AVF Control Number: NIST90NEC525_2_1.11
DATE COMPLETED
BEFORE ON-SITE: 1991-07-25
AFTER ON-SITE: 1991-09-18
REVISIONS: 1991-10-16

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 910918S1.11217
NEC Corporation
NEC Ada Compiler System for EWS-UX/V
to V70/RX-UX832, Version 1.0
EWS4800/60 => MV4000

Prepared By:
Software Standards Validation Group
Computer Systems Laboratory
National Institute of Standards and Technology
Building 225, Room A266
Gaithersburg, Maryland 20899

AVF Control Number: NIST90NEC525_2_1.11

Certificate Information

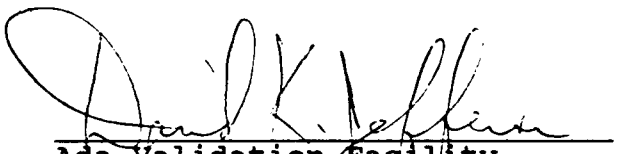
The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 18 SEPTEMBER 1991.

Compiler Name and Version:	NEC Ada Compiler System for EWS-UX/V to V70/RX-UX832, Version 1.0
Host Computer System:	EWS4800/60 running EWS-UX/V R8.1
Target Computer System:	MV4000 running RX-UX832 V1.6

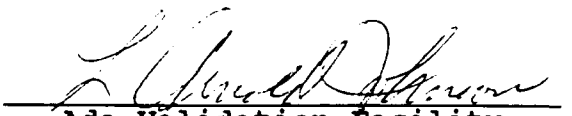
See section 3.1 for any additional information about the testing environment.

As a result of this validation effort, Validation Certificate 910918S1.11217 is awarded to NEC Corporation. This certificate expires on 01 March 1993.

This report has been reviewed and is approved.



Ada Validation Facility
Dr. David K. Jefferson
Chief, Information Systems
Engineering Division (ISED)



Ada Validation Facility
Mr. L. Arnold Johnson
Manager, Software Standards
Validation Group

Computer Systems Laboratory (CLS)
National Institute of Standards and Technology
Building 225, Room A266
Gaithersburg, MD 20899

Ada Validation Organization
Director, Computer & Software
Engineering Division
Institute for Defense Analyses
Alexandria VA 22311

Ada Joint Program Office
Dr. John Solomond
Director
Department of Defense
Washington DC 20301

AVF Control Number: NIST90NEC525_2_1.11

Certificate Information

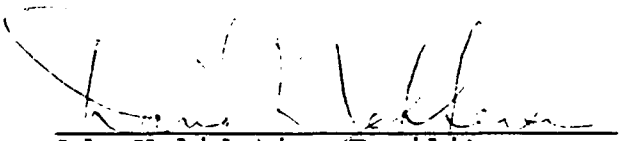
The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 18 SEPTEMBER 1991.

Compiler Name and Version:	NEC Ada Compiler System for EWS-UX/V to V70/RX-UX832, Version 1.0
Host Computer System:	EWS4800/60 running EWS-UX/V R8.1
Target Computer System:	MV4000 running RX-UX832 V1.6

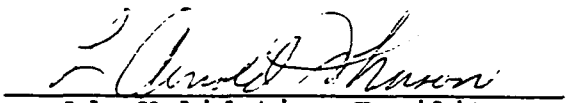
See section 3.1 for any additional information about the testing environment.

As a result of this validation effort, Validation Certificate 910918S1.11217 is awarded to NEC Corporation. This certificate expires on 01 March 1993.

This report has been reviewed and is approved.



Ada Validation Facility
Dr. David K. Jefferson
Chief, Information Systems
Engineering Division (ISED)



Ada Validation Facility
Mr. L. Arnold Johnson
Manager, Software Standards
Validation Group

Computer Systems Laboratory (CLS)
National Institute of Standards and Technology
Building 225, Room A266
Gaithersburg, MD 20899

Ada Validation Organization
Director, Computer & Software
Engineering Division
Institute for Defense Analyses
Alexandria VA 22311

Ada Joint Program Office
Dr. John Solomond
Director
Department of Defense
Washington DC 20301

NIST90NEC525_2_1.11

DECLARATION OF CONFORMANCE

The following declaration of conformance was supplied by the customer.

Customer: NEC Corporation
Igarashi Building
11-5, 2-CHOUME, SHIBAURA
Minato-ku, Tokyo, 108 Japan

Certificate Awardee: NEC Corporation
Igarashi Building
11-5, 2-CHOUME, SHIBAURA
Minato-ku, Tokyo, 108 Japan

Ada Validation Facility: National Institute of Standards and Technology
Computer Systems Laboratory (CSL)
Software Validation Group
Building 225, Room A266
Gaithersburg, Maryland 20899

ACVC Version: 1.11

Ada Implementation:

Compiler Name and Version: NEC Ada Compiler System for EWS-UX/V to V70/RX-UX832, Version 1.0

Host Computer System: EWS4800/60 running EWS-UX/V R8.1

Target Computer System: MV4000 running RX-UX832 V1.6

Declaration:

I the undersigned, declare that I have no knowledge of deliberate deviations from the Ada Language Standard ANSI/MIL-STD-1815A ISO 8652-1987 in the implementation listed above.

Tadashi Koizumi
Customer Signature and Certificate Awardee Signature

1991-09-05
Date

Mr. Tadashi Koizumi
Senior Manager
Environment Development Department
C&C Common Software Development Department
NEC Corporation

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

This Validation Summary Report (VSR) describes the extent to which a specific Ada compiler conforms to the Ada Standard, ANSI/MIL-STD-1815A. This report explains all technical terms used within it and thoroughly reports the results of testing this compiler using the Ada Compiler Validation Capability (ACVC). An Ada compiler must be implemented according to the Ada Standard, and any implementation-dependent features must conform to the requirements of the Ada Standard. The Ada Standard must be implemented in its entirety, and nothing can be implemented that is not in the Standard.

Even though all validated Ada compilers conform to the Ada Standard, it must be understood that some differences do exist between implementations. The Ada Standard permits some implementation dependencies--for example, the maximum length of identifiers or the maximum values of integer types. Other differences between compilers result from the characteristics of particular operating systems, hardware, or implementation strategies. All the dependencies observed during the process of testing this compiler are given in this report.

The information in this report is derived from the test results produced during validation testing. The validation process includes submitting a suite of standardized tests, the ACVC, as inputs to an Ada compiler and evaluating the results. The purpose of validating is to ensure conformity of the compiler to the Ada Standard by testing that the compiler properly implements legal language constructs and that it identifies and rejects illegal language constructs. The testing also identifies behavior that is implementation-dependent but is permitted by the Ada Standard. Six classes of tests are used. These tests are designed to perform checks at compile time, at link time, and during execution.

CHAPTER 1

INTRODUCTION

The Ada implementation described above was tested according to the Ada Validation Procedures [Pro90] against the Ada Standard [Ada83] using the current Ada Compiler Validation Capability (ACVC). This Validation Summary Report (VSR) gives an account of the testing of this Ada implementation. For any technical terms used in this report, the reader is referred to [Pro90]. A detailed description of the ACVC may be found in the current ACVC User's Guide [UG89].

1.1 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the Ada Certification Body may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject implementation has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from the AVF which performed this validation or from:

National Technical Information Service
5285 Port Royal Road
Springfield VA 22161

Questions regarding this report or the validation test results should be directed to the AVF which performed this validation or to:

Ada Validation Organization
Computer and Software Engineering Division
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311-1772

1.2 REFERENCES

[Ada83] Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

[Pro90] Ada Compiler Validation Procedures, Version 2.1, Ada Joint Program Office, August 1990.

[UG89] Ada Compiler Validation Capability User's Guide, 21 June 1989.

1.3 ACVC TEST CLASSES

Compliance of Ada implementations is tested by means of the ACVC. The ACVC contains a collection of test programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable. Class B and class L tests are expected to produce errors at compile time and link time, respectively.

The executable tests are written in a self-checking manner and produce a PASSED, FAILED, or NOT APPLICABLE message indicating the result when they are executed. Three Ada library units, the packages REPORT and SPPRT13, and the procedure CHECK_FILE are used for this purpose. The package REPORT also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The package SPPRT13 is used by many tests for Chapter 13 of the Ada Standard. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. If these units are not operating correctly, validation testing is discontinued.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that all violations of the Ada Standard are detected. Some of the class B tests contain legal Ada code which must not be flagged illegal by the compiler. This behavior is also verified.

Class L tests check that an Ada implementation correctly detects violation of the Ada Standard involving multiple, separately compiled units. Errors are expected at link time, and execution is attempted.

In some tests of the ACVC, certain macro strings have to be replaced by implementation-specific values -- for example, the largest integer. A list of the values used for this implementation is provided in Appendix A. In addition to these anticipated test modifications, additional changes may be required to remove unforeseen conflicts between the tests and implementation-dependent characteristics. The modifications required for this implementation are described in section 2.3.

For each Ada implementation, a customized test suite is produced by the AVF. This customization consists of making the modifications described in the preceding paragraph, removing withdrawn tests (see section 2.1) and, possibly some inapplicable tests (see Section 3.2 and [UG89]).

In order to pass an ACVC an Ada implementation must process each test of the customized test suite according to the Ada Standard.

1.4 DEFINITION OF TERMS

Ada Compiler	The software and any needed hardware that have to be added to a given host and target computer system to allow transformation of Ada programs into executable form and execution thereof.
Ada Compiler Validation Capability (ACVC)	The means for testing compliance of Ada implementations, Validation consisting of the test suite, the support programs, the ACVC Capability user's guide and the template for the validation summary (ACVC) report.
Ada Implementation	An Ada compiler with its host computer system and its target computer system.
Ada Joint Program (AJPO)	The part of the certification body which provides policy and guidance for the Ada certification Office system.
Ada Validation Facility (AVF)	The part of the certification body which carries out the procedures required to establish the compliance of an Ada implementation.
Ada Validation Organization (AVO)	The part of the certification body that provides technical guidance for operations of the Ada certification system.
Compliance of an Ada Implementation	The ability of the implementation to pass an ACVC version.
Computer System	A functional unit, consisting of one or more computers and associated software, that uses common storage for all or part of a program and also for all or part of the data necessary for the execution of the program; executes user-written or user-designated programs; performs user-designated data manipulation, including

	arithmetic operations and logic operations; and that can execute programs that modify themselves during execution. A computer system may be a stand-alone unit or may consist of several inter-connected units.
Conformity	Fulfillment by a product, process or service of all requirements specified.
Customer	An individual or corporate entity who enters into an agreement with an AVF which specifies the terms and conditions for AVF services (of any kind) to be performed.
Declaration of Conformance	A formal statement from a customer assuring that conformity is realized or attainable on the Ada implementation for which validation status is realized.
Host Computer System	A computer system where Ada source programs are transformed into executable form.
Inapplicable test	A test that contains one or more test objectives found to be irrelevant for the given Ada implementation.
ISO	International Organization for Standardization.
LRM	The Ada standard, or Language Reference Manual, published as ANSI/MIL-STD-1815A-1983 and ISO 8652-1987. Citations from the LRM take the form "<section>.<subsection>:<paragraph>."
Operating System	Software that controls the execution of programs and that provides services such as resource allocation, scheduling, input/output control, and data management. Usually, operating systems are predominantly software, but partial or complete hardware implementations are possible.
Target Computer System	A computer system where the executable form of Ada programs are executed.
Validated Ada Compiler	The compiler of a validated Ada implementation.
Validated Ada Implementation	An Ada implementation that has been validated successfully either by AVF testing or by registration [Pro90].

Validation	The process of checking the conformity of an Ada compiler to the Ada programming language and of issuing a certificate for this implementation.
Withdrawn test	A test found to be incorrect and not used in conformity testing. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains erroneous or illegal use of the Ada programming language.

CHAPTER 2

IMPLEMENTATION DEPENDENCIES

2.1 WITHDRAWN TESTS

Some tests are withdrawn by the AVO from the ACVC because they do not conform to the Ada Standard. The following 95 tests had been withdrawn by the Ada Validation Organization (AVO) at the time of validation testing. The rationale for withdrawing each test is available from either the AVO or the AVF. The publication date for this list of withdrawn tests is 91-08-02.

E28005C	B28006C	C32203A	C34006D	C35508I	C35508J
C35508M	C35508N	C35702A	C35702B	B41308B	C43004A
C45114A	C45346A	C45612A	C45612B	C45612C	C45651A
C46022A	B49008A	B49008B	A74006A	C74308A	B83022B
B83022H	B83025B	B83025D	B83026B	C83026A	C83041A
B85001L	C86001F	C94021A	C97116A	C98003B	BA2011A
CB7001A	CB7001B	CB7004A	CC1223A	BC1226A	CC1226B
BC3009B	BD1B02B	BD1B06A	AD1B08A	BD2A02A	CD2A21E
CD2A23E	CD2A32A	CD2A41A	CD2A41E	CD2A87A	CD2B15C
BD3006A	BD4008A	CD4022A	CD4022D	CD4024B	CD4024C
CD4024D	CD4031A	CD4051D	CD5111A	CD7004C	ED7005D
CD7005E	AD7006A	CD7006E	AD7201A	AD7201E	CD7204B
AD7206A	BD8002A	BD8004C	CD9005A	CD9005B	CDA201E
CE2107I	CE2117A	CE2117B	CE2119B	CE2205B	CE2405A
CE3111C	CE3116A	CE3118A	CE3411B	CE3412B	CE3607B
CE3607C	CE3607D	CE3812A	CE3814A	CE3902B	

2.2 INAPPLICABLE TESTS

A test is inapplicable if it contains test objectives which are irrelevant for a given Ada implementation. The inapplicability criteria for some tests are explained in documents issued by ISO and the AJPO known as Ada Commentaries and commonly referenced in the format AI-ddddd. For this implementation, the following tests were determined to be inapplicable for the reasons indicated; references to Ada Commentaries are included as appropriate.

The following 201 tests have floating-point type declarations requiring more digits than SYSTEM.MAX_DIGITS:

C24113L..Y (14 tests)	C35705L..Y (14 tests)
C35706L..Y (14 tests)	C35707L..Y (14 tests)
C35708L..Y (14 tests)	C35802L..Z (15 tests)

C45241L..Y (14 tests)	C45321L..Y (14 tests)
C45421L..Y (14 tests)	C45521L..Z (15 tests)
C45524L..Z (15 tests)	C45621L..Z (15 tests)
C45641L..Y (14 tests)	C46012L..Z (15 tests)

C24113I..K (3 TESTS) use a line length in the input file which exceeds 126 characters.

The following 20 tests check for the predefined type LONG_INTEGER; for this implementation, there is no such type:

C35404C	C45231C	C45304C	C45411C	C45412C
C45502C	C45503C	C45504C	C45504F	C45611C
C45613C	C45614C	C45631C	C45632C	B52004D
C55B07A	B55B09C	B86001W	C86006C	CD7101F

C35404D, C45231D, B86001X, C86006E, and CD7101G check for a predefined integer type with a name other than INTEGER, LONG_INTEGER, or SHORT_INTEGER; for this implementation, there is no such type.

C35713B, C45423B, B86001T, and C86006H check for the predefined type SHORT_FLOAT; for this implementation, there is no such type.

C35713D and B86001Z check for a predefined floating-point type with a name other than FLOAT, LONG_FLOAT, or SHORT_FLOAT; for this implementation, there is no such type.

C45531M..P and C45532M..P (8 tests) check fixed-point operations for types that require a SYSTEM.MAX_MANTISSA of 47 or greater; for this implementation, MAX_MANTISSA is less than 47.

C45624A..B (2 tests) check that the proper exception is raised if MACHINE_OVERFLOWS is FALSE for floating point types and the results of various floating-point operations lie outside the range of the base type; for this implementation, MACHINE_OVERFLOWS is TRUE.

C4A013B contains a static universal real expression that exceeds the range of this implementation's largest floating-point type; this expression is rejected by the compiler.

B86001Y uses the name of a predefined fixed-point type other than type DURATION; for this implementation, there is no such type.

C96005B uses values of type DURATION's base type that are outside the range of type DURATION; for this implementation, the ranges are the same.

CA2009C and CA2009F check whether a generic unit can be instantiated before its body (and any of its subunits) is compiled; this implementation creates a dependence on generic units as

allowed by AI-00408 and AI-00506 such that the compilation of the generic unit bodies makes the instantiating units obsolete. (See section 2.3.)

CD1009C uses a representation clause specifying a non-default size for a floating-point type.

CD2A84A, CD2A84E, CD2A84I..J (2 TESTS), AND CD2A84O use representation clauses specifying non-default sizes for access types.

The tests listed in the following table check that USE_ERROR is raised if the given file operations are not supported for the given combination of mode and access method; this implementation supports these operations.

Test	File Operation	Mode	File Access Method
CE2102E	CREATE	OUT_FILE	SEQUENTIAL_IO
CE2102F	CREATE	INOUT_FILE	DIRECT_IO
CE2102J	CREATE	OUT_FILE	DIRECT_IO
CE2102N	OPEN	IN_FILE	SEQUENTIAL_IO
CE2102O	RESET	IN_FILE	SEQUENTIAL_IO
CE2102P	OPEN	OUT_FILE	SEQUENTIAL_IO
CE2102Q	RESET	OUT_FILE	SEQUENTIAL_IO
CE2102R	OPEN	INOUT_FILE	DIRECT_IO
CE2102S	RESET	INOUT_FILE	DIRECT_IO
CE2102T	OPEN	IN_FILE	DIRECT_IO
CE2102U	RESET	IN_FILE	DIRECT_IO
CE2102V	OPEN	OUT_FILE	DIRECT_IO
CE2102W	RESET	OUT_FILE	DIRECT_IO
CE3102F	RESET	Any Mode	TEXT_IO
CE3102G	DELETE	--------	TEXT_IO
CE3102I	CREATE	OUT_FILE	TEXT_IO
CE3102J	OPEN	IN_FILE	TEXT_IO
CE3102K	OPEN	OUT_FILE	TEXT_IO

The tests listed in the following table check the given file operations for the given combination of mode and access method; this implementation does not support these operations.

Test	File Operation	Mode	File Access Method
CE2105A	CREATE	IN_FILE	SEQUENTIAL_IO
CE2105B	CREATE	IN_FILE	DIRECT_IO
CE3109A	CREATE	IN_FILE	TEXT_IO

CE2203A checks that WRITE raises USE_ERROR if the capacity of an external sequential file is exceeded; this implementation cannot restrict file capacity.

EE2401D checks whether read, write, set_index, index, size, and end_of_file are supported for direct files for an unconstrained array type. USE_ERROR was raised for direct create. The maximum element size supported for DIRECT_IO is 32K bits.

CE2403A checks that WRITE raises USE_ERROR if the capacity of an external direct file is exceeded; this implementation cannot restrict file capacity.

CE3111B and CE3115A associate multiple internal text files with the same external file and attempt to read from one file what was written to the other, which is assumed to be immediately available; this implementation buffers output. (See section 2.3.)

CE3304A checks that SET_LINE_LENGTH and SET_PAGE_LENGTH raise USE_ERROR if they specify an inappropriate value for the external file; there are no inappropriate values for this implementation.

CE3413B checks that PAGE raises LAYOUT_ERROR when the value of the page number exceeds COUNT'LAST; for this implementation, the value of COUNT'LAST is greater than 150000, making the checking of this objective impractical.

2.3 TEST MODIFICATIONS

Modifications (see section 1.3) were required for 78 tests.

The following tests were split into two or more tests because this implementation did not report the violations of the Ada Standard in the way expected by the original tests.

B22003A	B26001A	B26002A	B26005A	B28003A	B29001A	B33301B
B35101A	B37106A	B37301B	B37302A	B38003A	B38003B	B38009A
B38009B	B55A01A	B61001C	B61001F	B61001H	B61001I	B61001M
B61001R	B61001W	B67001H	B83A07A	B83A07B	B83A07C	B83E01C
B83E01D	B83E01E	B85001D	B85008D	B91001A	B91002A	B91002B
B91002C	B91002D	B91002E	B91002F	B91002G	B91002H	B91002I
B91002J	B91002K	B91002L	B95030A	B95061A	B95061F	B95061G
B95077A	B97103E	B97104G	BA1001A	BA1101B	BC1109A	BC1109C
BC1109D	BC1202A	BC1202F	BC1202G	BE2210A	BE2413A	

C83030C and C86007A were graded passed by Test Modification as directed by the AVO. These tests were modified by inserting "PRAGMA ELABORATE (REPORT);" before the package declarations at lines 13 and 11, respectively. Without the pragma, the packages may be elaborated prior to package Report's body, and thus the packages' calls to function REPORT.IDENT_INT at lines 14 and 13, respectively, will raise PROGRAM_ERROR.

C85006C was graded passed by Processing Modification as directed by the AVO. This test contains a task that is too large to be processed with the default task stack size of 2048. Therefore, a task stack size of 3300 was specified with the linker option "-D 3300.

CA2009C and CA2009F were graded inapplicable by Evaluation Modification as directed by the AVO. These tests contain instantiations of a generic unit prior to the compilation of that unit's body; as allowed by AI-00408 and AI-00506, the compilation of the generic unit bodies makes the compilation unit that contains the instantiations obsolete.

BC3204C and BC3205D were graded passed by Processing Modification as directed by the AVO. These tests check that instantiations of generic units with unconstrained types as generic actual parameters are illegal if the generic bodies contain uses of the types that require a constraint. However, the generic bodies are compiled after the units that contain the instantiations, and this implementation creates a dependence of the instantiating units on the generic units as allowed by AI-00408 and AI-00506 such that the compilation of the generic bodies makes the instantiating units obsolete--no errors are detected. The processing of these tests was modified by re-compiling the obsolete units; all intended errors were then detected by the compiler.

AD8011A, BD8001A, BD8003A, BD8004A, & BD8004B were graded passed by Test Modification as directed by the AVO. These tests are applicable to implementations that support package MACHINE_CODE: the Class A test checks that a machine-code procdure can be compiled and called; the Class B tests check that compilation units are illegal if various conditions for the use of code statements are violated. This implementation requires that the implementation-defined pragma ABSTRACT_ACODE_INSERTIONS(TRUE) be inserted into the declarative part of each procedure that contains code statements; without this pragma, the compiler rejects the units at their context clause for package MACHINE_CODE (which is the behavior expected for an inapplicable grade). This implementation requires the pragma as an enabling device for the particular "A-code" machine instructions, in anticipation of the intended future provision of a second type of machine-code instructions.

CE3111B and CE3115A were graded inapplicable by Evaluation Modification as directed by the AVO. The tests assume that output from one internal file is unbuffered and may be immediately read by another file that shares the same external file. This implementation raises END_ERROR on the attempts to read at lines 87 and 101, respectively.

CHAPTER 3

PROCESSING INFORMATION

3.1 TESTING ENVIRONMENT

The testing environment consisted of a host computer, a target computer and two auxiliary computers. The host computer and the Sun-3/260 auxiliary computer are connected by Ethernet. The target computer and the EWS4800/220 auxiliary computer are connected via 3.5 inch floppy disks and by a RS232C line. The configurations of these computers are described by the following:

Host Computer System:	EWS4800/60 running EWS-UX/V R8.1
Auxiliary Computer System:	Sun-3/260 running SunOS, Version 4.0.1
Target Computer System:	MV4000 running RX-UX832 V1.6
Auxiliary Computer System:	EWS4800/220 running EWS-UX/V (Rel4.0), Release 2.1
Communication link:	Ethernet, RS232C, 3.5 inch floppy disks

For technical and sales information about this Ada implementation, contact:

Dr. Toshio Miyachi
Environment System Department
C & C Common Software Development Laboratory
NEC Corporation
Shibaura 2-11-5, Minato-ku, Tokyo, 108 Japan
VOICE TELEPHONE: 81-3-5476-1107
FAX TELEPHONE: 81-3-5476-1113

Testing of this Ada implementation was conducted at the customer's site by a validation team from the AVF.

3.2 SUMMARY OF TEST RESULTS

An Ada Implementation passes a given ACVC version if it processes each test of the customized test suite in accordance with the Ada Programming Language Standard, whether the test is applicable or inapplicable; otherwise, the Ada Implementation fails the ACVC [Pro90].

For all processed tests (inapplicable and applicable), a result was obtained that conforms to the Ada Programming Language Standard.

The list of items below gives the number of ACVC tests in various

categories. All tests were processed, except those that were withdrawn because of test errors (item b; see section 2.1), those that require a floating-point precision that exceeds the implementation's maximum precision (item e; see section 2.2), and those that depend on the support of a file system -- if none is supported (item d). All tests passed, except those that are listed in sections 2.1 and 2.2 (counted in items b and f, below).

a)	Total Number of Applicable Tests	3791	
b)	Total Number of Withdrawn Tests	95	
c)	Processed Inapplicable Tests	284	
d)	Non-Processed I/O Tests	0	
e)	Non-Processed Floating-Point Precision Tests	0	
f)	Total Number of Inapplicable Tests	284	(c+d+e)
g)	Total Number of Tests for ACVC 1.11	4170	(a+b+f)

3.3 TEST EXECUTION

A magnetic tape containing the customized test suite (see section 1.3) was taken on-site by the validation team for processing. The customized test suite was loaded onto an auxiliary computer, the Sun-3/260, via a magnetic tape device. The customized test suite was then transferred from the auxiliary computer to the host computer, the EWS4800/60, using UNIX rsh and dd commands of the host computer via Ethernet.

The tests were compiled and linked on the host computer system. The executable images, generated on the host system, were transferred from the host system to the auxiliary computer system, EWS4800/220 via Ethernet. The executable images were then transferred from the EWS4800/220 to the target system via 3.5 inch floppy disks. The executable images were executed on the target system. The execution results were transferred from the target system back to the EWS4800/220 via RS232C communication line. The execution results were then transferred from the EWS4800/220 to the Sun-3/260 by Ethernet where they were then captured on magnetic tape. Commands were transferred from the EWS4800/220 to the target system via RS232C communication line.

Testing was performed using command scripts provided by the customer and reviewed by the validation team. See Appendix B for a complete listing of the processing options for this implementation. It also indicates the default options. The compiler options invoked both explicitly and implicity for validation testing during this test were:

-a (for A, C , D and E tests not listed below, and L

tests)

-L -a (for B, tests, support files and the following D and E tests:

D29002K	D64005E0M	D64005EA	D64005EB	D64005EC
D64005ED	D64005EE	D64005EF	D64005F0M	D64005FA
D64005FB	D64005FC	D64005FD	D64005FE	D64005FF
D64005FG	D64005FH	D64005FI	D64005FJ	E28002A
E28002D	E28005D	E28002B	E43211B	E43212B
E52103Y	EB4011A	EB4012A	EB4014A	ED1D04A
ED2A26A	ED2A56A	ED4021A	ED4022B	ED4022C
ED7002B	ED9002A)			

The linker options invoked explicitly for validation testing were:

-D 3300 -a (for C85006C)

-a (for all other tests)

Test output, compiler and linker listings, and job logs were captured on magnetic tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

APPENDIX A

MACRO PARAMETERS

This appendix contains the macro parameters used for customizing the ACVC. The meaning and purpose of these parameters are explained in [UG89]. The parameter values are presented in two tables. The first table lists the values that are defined in terms of the maximum input-line length, which is the value for $MAX_IN_LEN--also listed here. These values are expressed here as Ada string aggregates, where "V" represents the maximum input-line length.

Macro Parameter	Macro Value
$MAX_IN_LEN	<126> -- Value of V
$BIG_ID1	(1..V-1 => 'A', V => '1')
$BIG_ID2	(1..V-1 => 'A', V => '2')
$BIG_ID3	(1..V/2 => 'A') & '3' & (1..V-1-V/2 => 'A')
$BIG_ID4	(1..V/2 => 'A') & '4' & (1..V-1-V/2 => 'A')
$BIG_INT_LIT	(1..V-3 => '0') & "298"
$BIG_REAL_LIT	(1..V-5 => '0') & "690.0"
$BIG_STRING1	'"' & (1..V/2 => 'A') & '"'
$BIG_STRING2	'"' & (1..V-1-V/2 => 'A') & '1' & '"'
$BLANKS	(1..V-20 => ' ')
$MAX_LEN_INT_BASED_LITERAL	"2:" & (1..V-5 => '0') & "11:"
$MAX_LEN_REAL_BASED_LITERAL	"16:" & (1..V-7 => '0') & "F.E:"
$MAX_STRING_LITERAL	'"' & (1..V-2 => 'A') & '"'

The following table contains the values for the remaining macro parameters.

Macro Parameter	Macro Value
$ACC_SIZE	32
$ALIGNMENT	4
$COUNT_LAST	2147483647
$DEFAULT_MEM_SIZE	2_097_152
$DEFAULT_STOR_UNIT	8
$DEFAULT_SYS_NAME	V70_RXUX
$DELTA_DOC	2#1.0#E-31
$ENTRY_ADDRESS	16#40#
$ENTRY_ADDRESS1	16#80#
$ENTRY_ADDRESS2	16#100#
$FIELD_LAST	35
$FILE_TERMINATOR	' '
$FIXED_NAME	NO_SUCH_FIXED_TYPE
$FLOAT_NAME	NO_SUCH_TYPE
$FORM_STRING	""
$FORM_STRING2	"CANNOT_RESTRICT_FILE_CAPACITY"
$GREATER_THAN_DURATION	75_000.0
$GREATER_THAN_DURATION_BASE_LAST	131_073.0
$GREATER_THAN_FLOAT_BASE_LAST	1.80141E+38
$GREATER_THAN_FLOAT_SAFE_LARGE	1.0E308
$GREATER_THAN_SHORT_FLOAT_SAFE_LARGE	1.0E308
$HIGH_PRIORITY	255

```
$ILLEGAL_EXTERNAL_FILE_NAME1 /NODIRECTORY/FILENAME

$ILLEGAL_EXTERNAL_FILE_NAME2
                              "NAMES-OF-MORE-THAN-14-CHARACTERS" &
                              "NAMES-OF-MORE-THAN-14-CHARACTERS" &
                              "NAMES-OF-MORE-THAN-14-CHARACTERS" &
                              "NAMES-OF-MORE-THAN-14-CHARACTERS" &
                              "NAMES-OF-MORE-THAN-14-CHARACTERS" &
                              "NAMES-OF-MORE-THAN-14-CHARACTERS" &
                              "NAMES-OF-MORE-THAN-14-CHARACTERS" &
                              "NAMES-OF-MORE-THAN-14-CHARACTERS" &
                              "NAMES-OF-MORE-THAN-14-CHARACTERS"

$INAPPROPRIATE_LINE_LENGTH    -1

$INAPPROPRIATE_PAGE_LENGTH    -1

$INCLUDE_PRAGMA1              PRAGMA INCLUDE ("A28006D1.TST")

$INCLUDE_PRAGMA2              PRAGMA INCLUDE ("B28006E1.TST")

$INTEGER_FIRST                -2147483648

$INTEGER_LAST                 2147483647

$INTEGER_LAST_PLUS_1          2_147_483_648

$INTERFACE_LANGUAGE           AS, C

$LESS_THAN_DURATION           -75_000.0

$LESS_THAN_DURATION_BASE_FIRST  -131_073.0

$LINE_TERMINATOR              character'val(10)

$LOW_PRIORITY                 0

$MACHINE_CODE_STATEMENT
        AA_INSTR'(AA_EXIT_SUBPRGRM,0,0,0,AA_INSTR_INTG'FIRST,0);

$MACHINE_CODE_TYPE            AA_INSTR

$MANTISSA_DOC                 31

$MAX_DIGITS                   15

$MAX_INT                      2147483647

$MAX_INT_PLUS_1               2_147_483_648

$MIN_INT                      -2147483648
```

```
$NAME                       NO_SUCH_TYPE_AVAILABLE

$NAME_LIST                  V70_RXUX

$NAME_SPECIFICATION1        /X2120A

$NAME_SPECIFICATION2        /X2120B

$NAME_SPECIFICATION3        /X3119A

$NEG_BASED_INT              16#F000000E#

$NEW_MEM_SIZE               2_097_152

$NEW_STOR_UNIT              8

$NEW_SYS_NAME               V70_RXUX

$PAGE_TERMINATOR            ASCII.FF

$RECORD_DEFINITION          RECORD
                             instr_no : aa_instr_intg;
                             arg0     : aa_instr_intg;
                             arg1     : aa_instr_intg;
                             arg2     : aa_instr_intg;
                             arg3     : aa_instr_intg;
                             arg4     : aa_instr_intg;
                            END RECORD;

$RECORD_NAME                aa_instr

$TASK_SIZE                  32

$TASK_STORAGE_SIZE          2048

$TICK                       0.001

$VARIABLE_ADDRESS  system.address(16#d000_0000#-16#10000_0000#)

$VARIABLE_ADDRESS1 system.address(16#d000_0004#-16#10000_0000#)

$VARIABLE_ADDRESS2 system.address(16#d000_0008#-16#10000_0000#)

$YOUR_PRAGMA                PHYSICAL_ADDRESS
```

APPENDIX B

COMPILATION SYSTEM OPTIONS

The compiler options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to compiler documentation and not to this report.

NEC Ada Compilation System for EWS-UX/V to V70/RX-UX832

Compiler Options

1991

NEC Corporation

Contents

1 Options and Commands of The PLU

The Program Library Utility (PLU) of NEC Ada Compilation System for EWS-UX/V to V70/RX-UX832 is an interactive program that enables the user to manipulate libraries and their contents. Commands are provided for creating and deleting both root-libraries and sublibraries. Library contents such as directory information, and declarative information about exported symbols within compilation units can be displayed. Compilation units can be locked to prevent unwanted recompilation or deletion.

1.1 The Invocation Command

The PLU is invoked by entering the following commands to the shell:

```
apluv70 {<option>} [<unit-specifier>]
```

1. *Options*
   The following options are permitted:

   `-a < file-spec>`
   If this option is given, the PLU will attempt to open the sublibrary given by <file-spec>. If the qualifier is not present, the default is the sublibrary designated by the environment variable "ADA_LIBRARY". If the variable does not exist, the file ADA_LIBRARY is used.

   `-s LOCAL|GLOBAL`
   This option specifies the default library scope. The "LOCAL" scope allows operating only on currently open sublibrary. The "GLOBAL" scope allows operating on all sublibraries in the program library. The default scope is "LOCAL". Note that the default scope can be over-ridden on any individual commands through the use if explicit "SCOPE" option.

   `-c <string>`
   If present, this option causes the PLU to be invoked in a non-interactive mode, where the single command included in the <string> will be executed, after which the PLU will be exit. This option is convenient within shell scripts and background processes.

2. *Parameters*
   The PLU takes a single, optional parameter, that specifies the default compilation unit to be operated on. The <unit-specifier> can be any fully qualified (i.e. Ada syntax) compilation-unit name.

   To obtain information about a specific unit (e.g. the names and structure of the types exported by the unit) the PLU should be invoked with the name as a parameter. The PLU commands will then by default operate on that unit. This parameter is most useful in conjunction with the "LIST" command and its use will be described more fully in the section describing that command.

1.2 PLU Commands

Once invoked (unless the "-c" option is given), the PLU issues the prompt:

```
PLU>
```

after which the user can enter any PLU commands, terminated by a carriage return. The following PLU commands are currently available:

CREATE	– Creates a sublibrary, or root library.
DELETE	– Deletes the specified units or the current sublibrary.
DEPENDENCIES	– Shows the dependencies of the specified compilation units.
DIRECTORY	– Displays information about specified compilation units.
EXIT	– Exits the PLU session.
HELP	– Displays help information for the specified command.
LIBRARY	– Shows/changes the current program library.
LIST	– Displays symbolic information within specified units.
LOCK	– Locks the specified compilation units.
LOGGING	– Permits logging of a PLU session to a file.
QUIT	– Same as EXIT.

SCOPE	– Shows/changes the current default library scope.
SHOW	– Same as DIRECTORY.
TYPE	– Types source texts for the specified units.
UNIT	– Shows/changes the current default unit–name.
UNLOCK	– Unlocks the specified compilation units.
VERSION	– Shows the version of the library.

All commands, options and other keywords may be abbreviated to the shortest unique prefix required to identify them. In the following, this identification will be written in capital letters. All input to the PLU is case sensitive.

1.3 Unit Masks

Whenever a <unit_mask> is required (e.g. in the SHOW command), the name specified may include the wildcard characters "%" and "*" (representing any single character, or any (possibly empty) arbitrary string, respectively). These names may also be qualified by unit–name prefixes, separated by the "." character, as with the normal Ada subunit naming conventions. The first component of a <unit–mask> may be an integer, representing a unit–number. For example, the following are valid <unit–mask>'s:

```
23          -- The unit with unit number 23
23.JUNK *   -- All subunits of unit 23's JUNK subunit
UNIT_%%%    -- All units whose names are 8 characters long,
               the first 5 being "UNIT_"
*           -- All units
```

Each of the following sections describe the PLU commands in detail.

1.4 Symbol Masks

When using the list command, you may specify a symbol mask. Such a mask specifies entities in the usual Ada manner. A package named P is referred to as P, and any identifiers in it are referenced by the familiar dot notation.

For instance "TEXT_IO" denotes the package TEXT_IO, and "TEXT_IO.*" denotes all identifiers defined in TEXT_IO.

1.5 Command Summary

This section details the syntax and semantics of each PLU command.

1.5.1 CREATE

1. *Command syntax:*

```
Create <sublibrary-name> [<parent-library-name>]
```

   Creates a new sublibrary (or root library) in the file system.

2. *Options*

   (a) `-SIze = <number_of_blocks>`
   Specifies the number of blocks initially allocated for the sublibrary in 512 byte blocks. The size given here is only an initial size. The sublibrary will grow as required. If the size option is not given, 100 blocks are allocated by default.

   (b) `-ROot`
   Root specefies that the library created will be a copy of the system root sublibrary.

3. *Parameters*
   If two parameters are given, the second parameter must be the name of an existing sublibrary, which will be used as the parent. The first parameter is the name of the created (offspring) sublibrary.

   If only one parameter is given, the action taken depends on the presence of the "`-ROOT`" and "`-NEW_ROOT`" options.

   If neither of these (mutually exclusive) options are given, the created sublibrary will be the offspring of the currently open sublibrary. If one of the "`ROOT`" options is given, the sublibrary is created as a root sublibrary. In any case, the parameters given must be valid filenames.

4. *Example*

```
PLU> Create my_library.alb father.alb
```

This command will create an empty sublibrary, that will have the sublibrary father.alb as its parent.

1.5.2 DELETE

1. *Command syntax:*

```
DELete [<unit-mask>]
```

The specified library units are deleted from the current sublibrary (unless they are locked – see the "LOCK" command). Note that "–SCOPE" option or default scope is not appropriate for this command – it operates only on the current sublibrary (see the "LIBRARY" command).

2. *Options*

   (a) –CONFirm
   If this option is given, the PLU will prompt the user for confirmation before deleting each unit.

   (b) –BOdy
   If this option is given, only the applicable body units will be deleted.

   (c) –SPecification
   Both the specification and body units will be deleted.

   (d) –LIbrary
   This option indicates that the entire current sublibrary will be deleted (including all its units). The <unit–mask> parameter is not allowed in this option.

3. *Parameters*

The <unit−mask> is a qualified, Ada−format unit−name, potentially including wild−card characters (see note at end of section 1.3). If omitted, the default unit−mask will be used (see the "UNIT" command).

1.5.3 DEPENDENCIES

1. *Command syntax:*

```
DEPendecies [<unit-mask>]
```

Shows the dependencies recorded for the designated units.

2. *Options*

(a) −SPecification

(b) −BOdy
If neither of these options are specified, they are both active by default. If only one is specified, the other is inactive by default. The options specify the amount of information displayed:

- if "−SPECIFICATION" is given, dependencies recorded for the designated declaration units are displayed;
- if "−BODY" is given, dependencies recorded for the designated body units are displayed.

(c) −SCope = LOCAL | GLOBAL
This option determines the allowed libraries used by PLU in searching for the units specified by <unit−mask>. "LOCAL" allows searching in the current sublibrary, and "GLOBAL" allows searching in the entire current library. If −SCope is omitted, the current default scope will be used (see the "SCOPE" command).

3. *Parameters*
The <unit−mask> is a qualified, Ada−format unit−name, (see section 1.3) If omitted, the default unit−mask will be used (see the "UNIT" command).

1.5.4 DIRECTORY

1. *Command syntax:*

```
DIrectory [<unit-mask>]
```

Displays information about the contents of a sublibrary (i.e. its units).

2. *Synonyms:*
SHow

3. *Options*

(a) `-SCope = LOCAL | GLOBAL`
This option determines the allowed libraries used by the PLU in searching for the units specified by `<unit-mask>`. "`LOCAL`" allows searching in the current sublibrary, and "`GLOBAL`" allows searching in the entire current library. If the "`-SCOPE`" option is omitted, the current default scope will be used (see the "`SCOPE`" command).

(b) `-SUbunits`
If this option is given, the information displayed (recursively) includes any subunits that the designated units may have.

(c) `-DEPendencies`
Specifies that the information displayed shall include the dependecies recorded for the selected units.

(d) `-Unit_id`
If this option is given, the unit-id of each selected unit is displayed.

(e) `-ATtributes`
Specifies that information pertaining to the attributes of the selected units is to be included.

(f) `-BRief`
Gives a brief list of the specified unit. Only one line per units is given.

(g) `-SInce = <time>`
Only the units that are compiled after the given time are displayed.
The time format is that of the UNIX time specification, e.g. `May 3 12:30:21 1991`.
Data may be omitted from the right, so `Jul 3` and `Jul 3 12:30` are examples of valid time specifications. For the user's convenience, the pseudo time "`TODAY`" is valid, with the obvious semantics.

(h) `-CONTainer`
Specifies that the information displayed shall include the names of the containers assigned to the selected units.

(i) `-ALl`
If this option is given, all available information on the selected units is displayed (equivalent to giving all the above options except `BRIEF`).

4. *Parameters*

The <`unit-mask`> is a qualified, Ada-format `unit-name`, as described in section 1.3. If omitted, the default `unit-mask` will be used (see the "`UNIT`" command).

Both the specification and corresponding body of selected units will be displayed during a DIRECTORY listing.

1.5.5 EXIT

1. *Command syntax:*

```
EXIT
```

Closes the current library and the log file (if any), and terminates the session.

2. *Synonyms:*
Quit

1.5.6 HELP

1. *Command syntax:*

```
HELP [<command>]
```

Provides explanation of the PLU commands. If <command> is not specified, a brief summary of all available PLU commands will be displayed. Otherwise, the command syntax and applicable options for the specified command is shown.

1.5.7 LIBRARY

1. *Command syntax:*

```
LIBrary [<library-name>]
```

Selects a new default sublibrary, or displays the filenames of the sublibraries making up the current library.

2. *Parameters*

The <library−name> is a valid filename of the sublibraries making up the current library are displayed.

1.5.8 LIST

1. *Command syntax:*

```
LISt [<symbol-mask>]
```

This very powerful command displays the contents of the symbol−table for specified units. Using wildcards and appropriate options, the user can determine such information as the name of the unit where a particular declaration occurs, or a subroutine parameter profile.

2. *Options*

(a) −SCope = LOCAL | GLOBAL
This option determines where the PLU searches for symbols specified by tt <symbol−mask>. LOCAL restricts the search to the current sublibrary, GLOBAL searches the entire current library. If the "−SCOPE" option is omitted, the current default scope will be used (see the SCOPE command).
It is advisable to restrict the search space for the LIST command by using as few wildcards as possible, and by using local scope in order to minimize the the amount of output produced.

(b) −Entity_kind = (CONSTANT | VARIABLE | TYPE |
ENUMERATION_LITERAL | EXCEPTION | PROCEDURE |
FUNCTION | SUBPROGRAM | PACKAGE |
GENERIC | VALUE | PREDEFINED_OPERATOR | ALL)
Entity_kind can be used to limit the display to only those symbols matching the 'entity−kind' given. This is useful, both to limit the amount of information displayed, and to limit the search time. This option can also be used to reduce ambiguity when overloaded names are encountered. most of the entity−kinds correspond to Ada style terminology, but a few convenient terms have been added, such as VALUE (corresponding to CONSTANT, VARIABLE, ENUMERATION_LITERAL or FUNCTION). The special term PREDEFINED_OPERATOR displays the compiler generated operators that are created for each user −declared numeric type. Only explicitly declared operators are displayed by default.

(c) −LINe
The source−file line−number of each symbol's declaration will be displayed as a comment. Note that these line numbers are, in some cases, approximate.

(d) −DEClaration
An Ada−like declaration of the symbol will be displayed. This allows display of field names in record types, parameter profiles for subroutine declaration, and other useful information. The declaration is as much like a normal Ada declaration as possible, although certain kinds of information (such as the ordinal value of enumeration literals) are displayed in non−Ada form.

3. *Parameters* The tt <symbol−mask> is qualified, Ada_format symbol_name. The Ada notation may be used, both to specify subunits and local declarative items. If the <**symbol−mask**> begins with a "." character, then it must be proceeded by the default **unit−mask** (see the "UNIT" command). The following are examples of ways to specify the PUT procedures in FLOAT_IO:

```
        "TEXT_IO.FLOAT*.PUT"
".PUT", if default unit is TEXT_IO.FLOAT_IO
```

1.5.9 LOCK

1. *Command syntax:*

   LOCK [<unit-mask>]

   Locks the specified units such that they cannot be deleted or recompiled without being unlocked first (see the "UNLOCK" command). By default, this command locks the specification, the corresponding body and its subunits.

2. Options

   (a) **−CONFirm**
   Prompts for confirmation before locking each unit.

   (b) **−BOdy**
   The applicable specification and body units will be locked, but not the subunits.

   (c) **−SPecification**
   Only the specification units will be locked.

3. Parameters
   The < **unit−mask**> is a qualified, Ada−format unit−name, potentially including wild−card characters (see note at end of section 1.3).

1.5.10 LOGGING

1. *Command syntax:*

```
LOGging [<log-file-name>]
```

Allows output to be directed to a log file. If the <log_file_name> parameter is omitted, the current log file name will be displayed.

2. *Options*

(a) `-OFf`
Terminates logging, closing the current log−file. If this option is given, no parameter may be specified.

1.5.11 QUIT

1. *Command syntax:*

```
Quit
```

Quit is synonym for `EXIT` and pressing <`CTRL−D`>. Quit closes the current log file (if there is any), closes the current library, and terminates the PLU session.

1.5.12 SCOPE

1. *Command syntax:*

```
SCope [LOCAL | GLOBAL]
```

Allows setting of a default library scope for the various PLU commands such as the List command. If parameter is not given, the current default scope is displayed. The initial default scope is taken from the command invocation of the PLU.

2. *Parameters*
The parameter determines where the PLU looks for units specified by <`unit−mask`>. LOCAL means searching in the current sublibrary, and GLOBAL means searching in the current library.

1.5.13 SHOW

1. *Command syntax:*

```
SHow [<unit-mask>]
```

This i a synonym for DIrectory, q.v.

1.5.14 TYPE

1. *Command syntax:*

```
Type [<unit-mask>]
```

Allows listing of the source texts of the specified units. If the source–text for a specified unit is not contained in the library, the following message will be displayed:

```
-- No source code found for <unit-name>
```

By default, only the specifications of specified units are listed.

2. *Options*

   (a) `–LOg`
   Will cause the source–test listing to be directed to a file having the name `<unit–name>_s.ada` if a specification is being listed, of `<unit–name>_b.ada` if a body is being listed. Files will be written to the current default working directory.

   (b) `–Specification`
   Will cause the specifications of the designated units to be listed.

   (c) `–Body`
   Will cause the bodies of the designated units to be listed.

   (d) `–LIne`
   Causes line numbers to be placed in the left most columns of the listing.

(e) −SCope = LOCAL | GLOBAL
This option determines where the PLU looks for the units specified by <unit−mask>. "LOCAL" searches in the current sublibrary, and "GLOBAL" searches in the entire current library. If the "−SCOPE" option is omitted, the current default scope will be used (see the "SCOPE" command).

3. *Parameters*
The <unit−mask> is a qualified, Ada−format unit−name, as described in section 1.3. If omitted, the default unit−mask will be used (see the "UNIT" command).

1.5.15 UNIT

1. *Command syntax:*

```
UNIt [<unit-mask>]
```

Allows setting of a default unit−mask for those PLU commands that operate on units. If the parameter is omitted, the current default unit−mask is displayed. The initial default unit−mask is taken from the PLU invocation command line.

2. *Parameters*
The <unit-mask> is a qualified, Ada−format unit−name, as described in Section 1.3.

1.5.16 UNLOCK

1. *Command syntax:*

```
UNLock [<unit-mask>]
```

Unlocks the specified units so that they can be deleted or recompiled. By default, this command unlocks the specification, the corresponding body and all the body's subunits.

2. *Options*

(a) `-CONFirm`
If this option is given, the PLU will prompt for confirmation before unlocking each unit.

(b) `-Body`
If this option is given, only the body and subunits are unlocked.

(c) `-SUbunit`
Only the subunits are unlocked.

3. *Parameters*
The <`unit-mask`> is a qualified, Ada-format unit-name, as described in Section 1.3. If omitted, the default unit-mask will be used (see the "`UNIT`" command).

1.5.17 VERSION

1. *Command syntax:*

```
Version
```

Shows the version of the compiler which was used to compile the units in the current library.

2 Options for The Ada Compiler

The Ada Compiler is invoked by specifying a call of the program Ada to the shell. The invocation command is described in Section 2.1.

If any diagnostic messages are produced during the compilation, they are output on the diagnostic file and on the standard output . The diagnostic file and the diagnostic messages are described in Sections 2.1.3 and 2.3.

The user may request additional listings to be output on a list file by specifying options in the compiler invocation. The list file and the listings are described in Sections 2.1.2.

The compiler uses a program library during the compilation. The compilation unit may refer to units from the program and will be showed in the program library as a result of a successful compilation. The program library is described in the attachment. Section 2.4 briefly describes how the Ada compiler uses the library.

2.1 The Invocation Command

The invocation command has the following syntax:

```
adav70 <source-file-name> {<source-file-name>}
```

1. *Options*

   `-L and -l`
   Causes the compiler to produce a formatted listing of the input sources. The listing is written on the list file. Section 2.3.2 contains a description of the source listing. The default is no list file, in which case no source listing is produced, regardless of any LIST pragmas in the program or any diagnostic messages produced.

   `-x`
   Causes the compiler to produce a cross-reference listing. If this option is given and no severe or fatal errors are found during the compilation, the cross-reference listing will be written on the list file. The default excludes cross-reference.

   `-p`
   Progress-report.

   `-a <lib_spec>`
   Specifies the current sublibrary, and therefore the program library. If this option is omitted the sublibrary designated by the environment variable `ADA_LIBRARY` is used. If the variable does not exist the file `ADA_LIBRARY` is used. Section 2.4 describes how the Ada compiler uses the library.

   `-c <file_name>`
   Specifies the configuration file to be used by the compiler in the current compilation. If this option is omitted, the configuration file(config) in the compiler directory is used.

   `-s and -S`
   Specifies that the source text is not to be saved in the program library. This saves some space in the sublibrary. The default is to save source text. In this way, the user is always certain what

version of the source text was compiled. The source text may be displayed from the sublibrary with the PLU Type command.

`-B`

Build standard. Pseudo compilation of package standard. This option is intended for maintenance purposes only.

`-n`

No check. Suppress all run−time checks. By default, all run−time checks are generated.

`-N <keyword> {, <keyword>}`

Toggle check. Selective suppress of run−time checks. If a check is suppressed, the option will enable the check. If a check is enabled, the option will suppress the check. The following keywords are allowed:

- access
- index
- discriminant
- range
- length
- elaboration
- storage

Keywords are case−insensitive and can be abbreviated such that the abbreviation is unique.

`-o`

Optimize. Optimize the program with respect to execution time, which, under normal circumstances, also is optimization with respect to size of the executable.

`-u <unit_number>`

Specifies that the compilation unit being compiled is assigned the unit number<unit_number> in the current sublibrary (see the attachment for explanation of unit numbers). This option will only work for:

- compilations containing a single compilation unit which is neither a subunit nor contains subunit stubs,

- unit numbers which are unused and smaller than 32768.

2. *Parameters*
   The < `source-file-name`> specifies the file containing the source texts to be compiled. A source file is expected to have the string "`.ada`" as the last four characters of its name. If the last part of the name does not contain "." , the string "`.ada`" is appended to the name . More than one file name must be specified.

2.1.1 The List File

The name of the list file is identical to the name of the source file except that the final characters "`.ada`" are replaced by "`.lis`". The list file will be placed in the current directory.

2.1.2 The Diagnostic File

The name of diagnostic file is identical to the name of the source file except that the final characters "`.ada`" are replaced by "`.err`". The diagnostic file will be placed in the invoker's current directory.

The diagnostic file is a file containing a list of diagnostic messages, each followed by a line showing the number of the line in the source that caused the message to be generated, and then by a blank line. The file is not separated into pages and there are no headings.

2.1.3 The Configuration File

Certain functional characteristics of the compiler may be modified by the user. These characteristics are passed to the compiler by means of a configuration file, which is a text file. The contents of the configuration file must be an Ada positional aggregate, written on one line, of the type CONFIGURATION_RECORD, which is described below. The configuration file is not accepted by the compiler in the following cases:

- The syntax does conform with the syntax for positional Ada aggregates.
- A value is outside the ranges specified below.
- A value is not specified as a literal.

- LINES_PER_PAGE is not greater then `TOP_MARGIN + BOTTOM_MARGIN`.
- The aggregate occupies more than one line.

If the compiler is unable to accept the configuration file, an error message is written on the standard out put and the compilation is terminated.

```
Type OUTFORMATTING is
record
LINES_PER_PAGE   : INTEGER range 30..100;
-- cf. Section 2.3.1
TOP_MARGIN       : INTEGER range 4..90;
-- cf. Section 2.3.1
BOTTOM_MARGIN    : INTEGER range 0..90;
-- cf. Section 2.3.1
OUT_LINELENGTH   : INTEGER range 80..132;
-- cf. Section 2.3.1
SUPPRESS_ERRORNO : BOOLEAN;
-- cf. Section 2.3.4
end record;

Type INPUT_FORMATS is
(ASCII);
-- cf. Section 2.2

Type INFORMATTING is
record
INPUT_FORMAT     : INPUT_FORMATS;
-- cf. Section 2.2
INPUT_LINELENGTH : INTEGER range 70..250;
-- cf. Section 2.2
end record;

Type CONFIGURATION_RECORD is
record
IN_FORMAT   : INFORMATTING;
OUT_FORMAT  : OUTFORMATTING;
ERROR_LIMIT : INTEGER;
```

```
-- cf. Section 2.3.4
end record;
```

The configuration file has the following content:

```
((ASCII, 126), (48, 5, 3, 100, FALSE), 200)
```

The name of this configuration file is passed to the compiler through the argument supplied with the -c option.

The output formatting parameters have the following meaning:

`LINES_PER_PAGE`:
: specifies the maximum number of lines written on each page (including top and bottom margin).

`TOP_MARGIN`:
: specifies the number of lines on top of each page used for a standard heading and blank lines. The heading is placed in the middle lines of the top margin.

`BOTTOM_MARGIN`:
: specifies the minimum number of lines left blank in the bottom of the page. The number of lines available for the listing of the program is LINES_PER_PAGE – TOP_MARGIN – BOTTOM_MARGIN.

`OUT_LINELENGTH`:
: specifies the maximum number of characters written on each line. Lines longer than OUT_LINELENGTH are separated into two lines.

`SUPPRESS_ERRORNO`:
: specifies the format of error messages.

2.2 The Source Text

The user submits one file containing a source text in each compilation. The source text may consist of one or more compilation units.

The format of the source text specified in the configuration file (cf. Section 2.1.3) must be the ISO-FORMAT ASCII. This format requires that the

source text is a sequence of ISO characters (ISO standard 646), where each line is terminated by either one of the following termination sequences (CR means carriage return, VT means vertical tabulation, LF means line feed, and FF means form feed):

- A sequence of one or more CRs, where the sequence is neither immediately preceded nor immediately followed by any of the characters VT, LF, or FF.
- Any of the characters VT, LF, or FF, immediately preceded and followed by a sequence of zero or more CRs.

In general, ISO control characters are not permitted in the source text with the following exceptions:

- The horizontal tabulation (HT) character may be used as a separator between lexical units.
- LF, VT, FF, and CR may be used to terminate lines, as described above.

The maximum number of characters in an input line is determined by the contents of the configuration file, cf. Section 2.1.3. The control characters CR, VT, LF, and FF are not considered a part of the line. Lines containing more than the maximum number of characters are truncated, and an error message is issued.

2.3 Compiler Output

The compiler may produce output in the list file, the diagnostic file and the standard output. It furthermore updates the program library if the compilation is successful. The present section describes the text output in the three files mentioned above. The updating of the program library is described in section 2.4.

The compiler may produce the following text output:

1. A listing of the source text with embedded diagnostic messages is written on the list file if the -L option is present.
2. A compilation summary is written in the list file if the -L option is present.

3. A cross-reference listing is written on the list file if the -x option is present and severe or fatal errors have been detected during the compilation.

4. If there are any diagnostic messages, a diagnostic file containing the diagnostic messages is written.

5. Diagnostic messages other than warnings are written on the standard output.

2.3.1 Format of the List File

The list file may include one or more of the following parts: a source listing, a cross-reference listing and a compilation summary.

The parts of the list file are separated by page ejects. The contents of each part are described in sections 2.3.2. - 2.3.3.

The format of the output on the list file is controlled by the configuration file (cf. Section 2.1.3) and may therefore be controlled by the user.

2.3.2 Source Listing

A source listing an unmodified copy of (pars of) the source text. The listing is divided into pages and each line is supplied with a line number.

The number of lines output in the source listing is governed by the occurrence of LIST pragmas and the number of objectionable lines.

- Parts of the listing can be suppressed by the use of LIST pragmas, and page breaks may be introduced by PAGE pragmas.

- A line containing a construct that caused a diagnostic message is printed even if it occurs at a point where the listing has been suppressed by a LIST pragma.

2.3.3 Compilation Summary

At the end of a compilation, the compiler produces a summary that is an output on the list file if the -L option is present.

The summary contains information about:

1. The type and name of the compilation unit, and whether it has been compiled successfully or not.
2. The number of diagnostic messages produced for each class of severity, cf Section 2.3.4.
3. Which options were present.
4. The full name of the source file.
5. The full name of the program library.
6. The number of source text lines.
7. The size of the text segment, the data segment, and the BSS segment.
8. Elapsed time and CPU time.
9. A "Compilation terminated" message if the compilation unit was the last in the compilation or "Compilation of next unit initiated" otherwise.

2.3.4 Diagnostic Messages

The Ada compiler issues diagnostic messages on the diagnostic file (cf. Section 2.1.2). Diagnostics other than warnings also appear on the standard output. If a source text listing is required, the diagnostics are also found embedded in the list file (cf. Section 2.1.1).

In a source listing , a diagnostic message is placed immediately after the source line causing the message. Messages not related to any particular line are placed at the top of the listing. Every diagnostic message in the diagnostic file is followed by a line stating the line number of the objectional line. The lines are ordered by increasing source line numbers. Messages not related to any particular line are assigned to line 0. On the standard output the messages appear in the order in which they are generated by the compiler.

The diagnostic messages are classified according to their severity and the compiler action taken:

Warning: Reports a questionable construct or an error that does not influence the meaning of the program. Warnings do not hinder the generation of object code.
Example: A warning will be issued for constructs for which the compiler detects that they will raise CONSTRAINT_ERROR at run time.

Error: Reports an illegal construct in the source program. Compilation continues, but no object code will be generated.
Example: most syntax errors; most static semantic errors.

Severe error: Reports an error which causes the compilation to be terminated immediately. No object code is generated. Example: A severe error message will be issued if a library unit mentioned by a WITH clause is not present in the current program library.

Fatal error: Reports an error in the compiler system itself. The compilation is terminated immediately and no object code is produced. The user may be able to circumvent a fatal error by correcting the program or by replacing program constructs with alternatives. Please inform NEC about the occurrence of fatal errors.

The detection of more errors than allowed by the number specified by the ERROR_LIMIT parameter of the configuration file (cf. Section 2.1.3) is considered a severe error.

2.4 The Program Library

This section briefly describes how the Ada compiler changes the program library. For a general general description of the program library refer to the attachment.

The compiler is allowed to read from all sublibraries constituting the current program library, but only the current sublibrary may be changed.

2.4.1 Correct Compilations

In this section, it is assumed that the compilation units are correctly compiled, i.e. that no errors are detected by the compiler.

Compilation of a Library Unit Which Is a Declaration

If a declaration unit of the same name as the one currently being compiled exists in the current sublibrary, it is deleted together with its body unit and the body's possible subunits. A new declaration unit is inserted in the sublibrary.

Compilation of a Library Unit Which Is a Subprogram Body

A subprogram body in a compilation unit is treated as a secondary unit if the current sublibrary contains:

- a valid subprogram declaration of the same name, or
- a valid generic subprogram declaration of the same name.

In all other cases, it will be treated as a library unit, i.e.:

- when there is no library unit of that name,
- when there is an invalid declaration unit of that name,
- when there is a package declaration, generic package declaration or an instantiated package or subprogram of that name.

Compilation of a Library Unit Which Is an Instantiation

A declaration unit with the name of the compilation unit in the current sublibrary is deleted (if it exits) with its body unit and possible subunits. A new declaration unit is inserted.

Compilation of a Secondary Unit Which Is a Library Unit Body

The existing body is deleted from the sublibrary together with its possible subunits. A new body unit is inserted.

Compilation of a Secondary Unit Which Is a Subunit

If the subunit exists in the sublibrary, it is deleted with its possible subunits. A new subunit is inserted.

2.4.2 Incorrect Compilations

If the compiler detects an error in a compilation, the program library will remain unchanged.

Note that if a file consists of several compilation units and an error is detected in any of these compilation units, the program library will not be updated for any of the compilation units.

LINKER OPTIONS

The linker options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to linker documentation and not to this report.

3 Commands of The Ada Linker

Before a compiled Ada program can be executed, it must be linked by the Ada Linker.

The Ada Linker performs two different jobs:

- It links an Ada program. The linker will check the consistency of the program (cf. section 2.3) and decide an elaboration order for the units constituting the program. Any errors found will be reported on the standard output and optionally on a log file. If no errors are found, an executable image file will be produced.

- It examines the consequences of recompilations. The linker will check the consistency of the specified program units (cf. section 2.3) as if the specified recompilations were actually performed, and determine an elaboration order for the program. Any errors found will be reported on the standard output and optionally on a log file, together with a list of needed recompilations.

It is possible to check the consequences of no recompilations, in which case the linker will check the consistency of the specified program units as they appear in the current program library.

3.1 The Invocation Command

The linker is invoked by submitting the following command to the command language interpreter:

```
alv70 {<options>} <unit_name>
```

<unit_name>

If a linking is requested, <unit_name> must specify a main program which is a library unit of the current program library, but not necessarily of current sublibrary. The library unit must be a parameterless procedure. Wildcard characters are not permitted.

If examination of the consequences of recompilations is requested, <unit_name> specifies a set of program library units whose consistency after the hypothetical recompilations will be checked. <unit_name> may contain wildcard characters which will be interpreted according to the rules of the UNIX in the current sublibrary with names that match specified <unit_name>. All types of library unit may be designated.

1. *Options*

   `-l <file_name>`
   Causes the linker to produce a log file named <file_name>. The log file is written to the invoker's current directory.

   `-L`
   Causes the linker to produce a log file named "link.log". The log file is written to the invoker's current directory.

   `-a <lib_spec>`
   Specifies the current sublibrary and thereby also the current program library. If this option is omitted then is sublibrary designated by the environment variable ADA_LIBRARY is used. If the environmental variable is not denied, the file ADA_LIBRARY is used.

   `-c`
   causes the linker to check the the consistency of the program(s) specified by <unit_name>. If this option is omitted then the specified program will be linked.

   `-s <unit_set> and -b <unit_set>`
   Define those library units after whose hypothetical recompilations the consistency will be checked. The meaning of <unit_set> is:
   <unit_set> ::= <unit_name> {, <unit_name>}
   For those unit names that appear with the -s option, the specifications of the corresponding library units are included in the list of hypothetical recompilations.
   For those unit names that appear with the -b option, the bodies of the corresponding library units are included in the list of hypothetical recompilations.
   A <unit_name> may appear in the <unit_set> of both options if required.

`-T <test_code>`
Causes the linker to send test output to the log file. The <`test_code`> is an integer in the range 0..9. This option is only allowed if the -l or -L option is present.

`-o <filename>`
Include the file denoted by <`filename`> in the link. Useful for including entities written in assembly language or C.

`-p "string"`
Options. The string will be inserted immediately after the "`ld`" command when the native linker is invoked. Useful for supplying options for the linker.

`-D <stack>`
Default `stack-size`. specifies the amount of stack allocated for task of a task type for which a length clause is not specified.

2. *Examples*

```
$ alv70 -a mylib  myprog
```

The linker will generate an executable image in the current directory from the program from the library mylib.

```
$ alv70 -L -c -s example -b utility myprog
```

This will examine the consequences of the recompilations of the example specification and the utility body. The linker will give a list of necessary compilations to keep myprog consistent. The program library is given by the environment variable `ADA_LIBRARY`.

```
$ alv70 -c -s a+b -b c prog_x
```

Here the linker will examine the consistency of the program "`prog_x`" in case of a recompilations of the specifications of the library unit "`a`" and all library units with names starting with "`b`", and a recompilation of the bodies of the all library units with names starting "`c`".

```
$ alv70 -o ObjectFile.o myprogram
```

The linker will include the code in `ObjectFile.o` in the linkage of the program "myprogram". This program presumably features a program interface to some routines appearing in the ObjectFile.

4 Options for The Ada System generator

asgv70 is a system that generates a load module of RX-UX832 from a SG definition file, object module files generated by alv70.

4.1 The Invocation Command

The Invocation command has the following syntax.

```
asgv70 {<option>} <SG-definition-file-name>
```

1. *Options*
   The following options are permitted:

   `-a <library-spec>`
   Specifies default Ada libraries.This option is used for a program for which ada_lib is not specified in SG definition file. This optiojn specifies the value of ada_lib in SG definition.

   `-conv`
   Convert endians of the load module form big(host cpu) to little(target cpu).

   `-h and -H`
   Displays messages for options.

   `-int <interrupt_mask>[, <interrupt_mask> ...]`
   Specifies interrupt_masks permitted by the task that is activated first. Notation of a interrupt_mask is a numeric literal notation of Ada (eg. 16#ff0#).

   `-i <file_name>`
   This option is used for a program for which obj_module is NOT specified in SG definition file. This option specifies the value of obj_module in SG definition file.

`-m <program_name>`
This option is used for a program for which main_program is NOT specified in SG definition file. This option specifies the value of main_program in SG definition file.

`-o <file_name>`
Specifies the name of load module generated by asgv70. If the name is omitted, the load module name is vsld.out.

`-P`
Specifies that temporary files generated by asgv70 are not removed, but are moved to the directory specified by -ts option.

`-R [<file_name>]`
Specifies that application load modules and the kernel load module are generated in different files. Specified file name is a kernel load module name. If a file_name is omitted, a kernel load module name is kernlm.

`-RA`
Specifies that only application load modules are generated.

`-RK [<file_name>]`
Specifies that only a kernel load module are generated. Specified file name is a kernel load module name. If a file_name is omitted, the kernel load module name is kernlm.

`-t <directory_name>`
Specifies a directory for temporary files generated by asgv70 and vsld commands. If a directory_name is omitted, temporary files for asgv70 are generated in the current directory, and temporary files for vsld are generated in /tmp directory.

`-tl [<file_name>]`
Specifies the name of a task library file generated by vsld command. If the file_name is not specified, the task library file name is a name of SG definition file following ".l". Without this option, a task library file generated by vsld command is removed.

`-ts <directory_name>`
Specifies the name of a directory for temporary files generated by asgv70.

-tv <directory_name>
Specifies the name of a directory for temporary files generated by vsld commands of RX-UX832 that is called in asgv70.

-v
Displays the RX-UX832 commands called by asgv70.

-V
Displays the version of asgv70. No load module is generated

APPENDIX C

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in Chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this Appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD, which are not a part of Appendix F, are:

```
package STANDARD is

type SHORT_INTEGER is range -32_768..32_767;

type INTEGER is range -2_147_483_648..2_147_483_647;

type FLOAT is digits 6
    range -16#0.FFFF_FF#E32..16#0.FFFF_FF#E32;

type LONG_FLOAT is digits 15
    range -16#0.FFFF_FFFF_FFFF_F8#E256..16#0.FFFF_FFFF_FFFF_F8#E256;
type DURATION is delta 2#1.0#E-14 range -131_072.0..131_071.0;

end STANDARD;
```

NEC Ada Compilation System for EWS-UX/V to V70/RX-UX832

Appendix F

1991

NEC Corporation

Contents

1 Introduction

This appendix describes the implementation-dependent characteristics of the NEC Ada Compilation System for EWS-UX/V to V70/RX-UX832 as required in the Appendix F frame of the Ada Reference Manual(ARM, ANSI/MIL-STD 1815A).

2 Implementation-Dependent Pragmas

This section lists the language defined pragmas, any restrictions on their use, and their effect compared to ARM explanation. This section also lists implementation–dependent pragmas.

2.1 Language Defined Pragmas

CONTROLLED

This pragma has no effect. No automatic reclaiming of storage is performed.

ELABORATE

As in ARM.

INLINE

Pragma INLINE causes inline expansion except in the following cases:

1. The whole body of the subprogram for which inline expansion is wanted has not been seen. This ensures that recursive procedures cannot be inline expanded.
2. The subprogram call appears in an expression on which conformance check may be applied, i.e. in a subprogram specification, in a discriminant part, or in a formal part of an entry declaration or accept statement.
   See the following example:

```
1 Package inline_test is
2
3 function one return integer;
```

```
 4 pragma inline (one);
 5
 6 end inline_test;
 7
 8 package body inline_test is
 9
10 function one return integer is
11 begin
12 return 1;
13 end one;
14
15 procedure def_parms (parm : integer := one) is
*** 46W-0: Warning : Inline expansion of ONE is not achieved
          here
16 begin
17 null;
18 end def_parms;
19
20 end inline_test;
21
```

3. The subprogram is an instantiation of the predefined generic subprograms `UNCHECKED_CONVERSION` or `UNCHECKED_DEALLOCATION`.
4. The subprogram is declared in a generic unit. The body of that generic unit is compiled as a secondary unit in the same compilation as a unit containing a call to (an instance of) the subprogram. See the following example:

```
 1 -- A compilation with there units:
 2 generic
 3 package g is
 4 procedure P;
 5 pragma inline( p );
 6 end g;
 7
 8 package body g is
 9 procedure p is
10 begin
```

```
   11 null;
   12 end p;
   13 end g;
   14
   15 with g;
   16 procedure example is
   17 package n is new g;
   18 begin
   19 n.p;
*** 43W-0: Warning Inline expansion of P is not achieved here

   20 end example;
```

5. The subprogram is declared by a renaming declaration.
6. The subprogram is passed as a generic actual parameter.

A warning is given if inline expansion is not achieved.

INTERFACE

Pragma INTERFACE is supported with C and V70 assembly language (AS) as target languages.

The following restrictions exist:

- The name of an interfaced subprogram must not have more that 30 characters. If name is over 30 characters, it is truncated.
- Case is significant in C, but not ion Ada. This conflict is resolved by matching the Ada name of an interfaced subprogram with a lower_case version of the name in C and prefix with a "_".
- User-declared name starting with the string "_ada".

The following rules exist for the matching of Ada types with C types:

- the following non-composite Ada type map directly onto an equivalent C type.

Ada	C
SHORT_INTEGER	short
INTEGER	int, long
enumeration	int
CHARACTER	int
BOOLEAN	int
FLOAT	float
LONG_FLOAT	double

When occurring as parameters of mode `IN`, objects of these type are assumed passed by value; when used as parameters of mode `IN OUT`or `OUT` they are assumed to be passed by address. Ada functions returning such types do so by value.

- Objects of record types are passed by address, regardless of size, for all parameter modes and for function return values. Only constrained record types can be interfaced.
- For objects of array types, the address of a data area alone is passed, regardless of dimensions, bounds or mode. Unconstrained array types may not be used as parameters of mode `IN OUT` or `OUT`, or as function return values.
- Ada unpacked arrays of type CHARACTER interface with C type (int *) . Ada objects of type STRING interface with C type (char *). It should be remembered that type STRING is packed (see ARM, Appendix C/17).
- Fixed types have no natural counterpart in C and cannot be interfaced.

LIST

As in ARM.

MEMORY_SIZE

Not supported, cf. Pragma SYSTEM_NAME.

OPTIMIZE

This pragma has no effect.

PACK

This pragma has the following form:

```
Pragma PACK (type_simple_name)
```

It is legal to specify this pragma for array and record types. It has no effect on record objects. The array components must be of a discrete type, i.e. integer, short_integer, and enumeration types (including boolean and character).

The pragma has the effect that object of the given type are packed into the nearest 2**n bits large enough to accomodate the object. For example, an object with a size of 3 bits is packed into 4 bits.

PAGE

As in ARM.

PRIORITY

As in ARM.

SHARED

Not allowed for variables of type LONG_FLOAT or subtypes or derived types thereof.

STORAGE_UNIT

Has no effect.

SUPPRESS

The implementation only supports the following form of the pragma:

```
Pragma SUPPRESS (identifier);
```

where identifier is one of the identifiers defined in ARM, Section 8.7, i.e. it is not possible to restrict the omission of a certain check to a specified name.

SYSTEM_NAME

Not supported. The only meaningful `SYSTEM_NAME` is `EWS_UVX_R4` when using the `EWS_UXV_R4` version of the Ada Compiler.

2.2 Implementation Defined Pragmas

INTERFACE_SPELLING

This pragma allows the user to call interfaced subprograms that have names that cannot be given to Ada subprograms because of Ada's lexical rules. Pragma interface_spelling supplies a mapping between the name of an Ada subprogram (that is mentioned in a pragma interface) and the name under which the subprogram is known in the surroundings.

Example

```
function Allocate (seize: integer) return byte_ptr; pragma interface
-- Supply the name of the c allocate function
pragma interface_spelling (Alloc,  "alloc");
```

If no pragma interface_spelling is given for an interfaced subprogram, the result will be as if a pragma interface_spelling had been given with a string containing the Ada spelling of the subprogram in lower case:

```
pragma interface (C,  Some_function);

-- Implicit pragma interface_spelling
   ( some_function,  "some_function");

pragma interface (AS,  Some_function)

-- Implicit pragma interface_spelling
   (some_function,  "SOME_FUNCTION")
```

ABSTRACT_ACODE_INSERTIONS

This pragma enables code statements in abstract acode.

CONCURRNCY

This pragma can appeare in the declarative part of a task and a library subprogram. It has one integer argument which limits concurrency of the task type in the system (not in a program). A concurrency is

associated with the main program if this pragma appears in its outer most declarative part. This pragma has no effect if it occures in a subprogram other than the main program.

INTERRUPT_MASK

This pragma can appear in the declarative part of a task and a library subprogram. It has no arguments. If it appears, no interrupts are accepted while the task is executing. A interrupt mask is associated with the main program if this pragma appears in its outer most declarative part. This pragma has no effect if it occures in a subprogram other than the main program.

TIME_SLICE

This pragma can appear in the declarative part of a task and a library subprogram. It has one integer argument which specifies time slice value of the task type in miliseconds. A time slice is associated with the main program if this pragma appears in its outer most declarative part. This pragma has no effect if it occures in a subprogram other than the main program.

DATA_PRESERVE

This pragma has no argument. If it appears in the outer most declarative part of main program, a level C restart of the operating system does not initialize statically allocated data. This pragma has no effect if it occures in a subprogram other than the main program.

PHYSICAL_ADDRESS

This pragma has one arguemnt, variable name which is also a specified address clause in the same declarative part. If this pragma is specified, the address of the address clause denotes a physical address rather than a virtual address. In this case physical memory is mapped to virtual space and the variable is accessed via the mapping. Because an address attribute of a variable denotes a virtual address of the variable, the attribute does not confirm the address of the address clause.

ROM

This pragma has one arguemnt, a variable name, which is also a specified address clause in the same declarative part. This pragma implies

a pragma physical_address. If this pragma is specified, its physical memory is treated as a ROM device.

RAM

This pragma has one arguemnt, a variable name, which is also a specified address clause in the same declarative part. This pragma implies a pragma physical_address. If this pragma is specified, its physical memory is treated as a RAM device and 0 cleared at the boot time.

IO_SPACE

This pragma has one arguemnt, a variable name, which is also a specified address clause in the same declarative part. This pragma implies a pragma physical_address. If this pragma is specified, its physical address is treated as an I/O address. The variable is mapped to the I/O space of V70 rather than memory space.

INTERFACE_LOCAL

This pramga has one argument, a subprogram name which is also specified in pragma interface. If this pragma is specified, the object file of external subprogram is assumed to be linked to each task. Statically allocated data in the object file has different addresses for each tasks.

3 Implementation-dependent Attributes

No implementation-dependent attributes are defined.

4 Package SYSTEM

```
package SYSTEM is

   type ADDRESS            is new  INTEGER;
   subtype PRIORITY        is INTEGER range 0 .. 255;
   type NAME               is ( V70_RXUX );
   SYSTEM_NAME:            constant NAME  := V70_RXUX;
   STORAGE_UNIT:           constant       := 8;
```

```
   MEMORY_SIZE:            constant        := 2048 * 1024;
   MIN_INT:                constant        := -2_147_483_648;
   MAX_INT:                constant        := 2_147_483_647;
   MAX_DIGITS:             constant        := 15;
   MAX_MANTISSA:           constant        := 31;

   FINE_DELTA:             constant        := 2#1.0#E-31;
   TICK:                   constant        := 0.001;

   type interface_language is (C,AS);

-- Compiler system dependent types:

    subtype INTEGER_16  is SHORT_INTEGER;
    subtype NATURAL_16  is INTEGER_16 range 0..INTEGER_16'LAST;
    subtype POSITIVE_16 is INTEGER_16 range 1..INTEGER_16'LAST;

    subtype INTEGER_32  is INTEGER;
    subtype NATURAL_32  is INTEGER_32 range 0..INTEGER_32'LAST;
    subtype POSITIVE_32 is INTEGER_32 range 1..INTEGER_32'LAST;
```

5 Representation Clauses

The representation clauses that are accepted are described below. Note that representation specification can be given on derived types too.

5.1 Pragma PACK

Pragma PACK applied on an array type will pack each array element into the smallest number of bits possible, assuming that the component type is a discrete type other than LONG_INTEGER or a fixed point type. Packing of arrays having other kinds of component types have no effect.

When the smallest number of bits needed to hold any value of a type is calculated, the range of the types is extended to include zero.

Pragma PACK applied on a record type will attempt to pack the components not already covered by a representation clause(perhaps none). This packing will begin with the small scalar components and larger components will follow in the order specified in the record. The packing begins at the first storage unit after the components with representation clauses.

The component types in question are the ones defined above for array types.

5.2 Length Clauses

Four kinds of length clauses are accepted.

1. *Size specifications:*

   The size attribute for a type T is accepted in the following cases:

   (a) If T is a discrete type, then the specified size must be greater than or equal to the number of bits needed to represent a value of the type, and less than or equal to 32. Note that when the number of bits needed to hold any value of the type is calculated, the range is extended to include 0 if necessary, i.e. the range 3..4 *cannot* be represented in 1 bit, but needs 3 bits.

   (b) If T is a fixed point type, then the specified size must be greater than or equal to the smallest number of bits needed to hold any value of the fixed point type, and less than 32 bits. Note that the Ada Reference Manual permits a representation, where the lower bound and the upper bound is not representable in the type. Thus the type

   ```
   type FIX is delta 1.0 range -1.0 .. 7.0;
   ```

   is representable in 3 bits. As for discrete types, the number of bits needed for a fixed point type is calculated using the range of the fixed point type possibly extended to include 0.0.

   (c) If T is a floating point type, an access type or task type the specified size must be equal to the number of bits used to represent values of the type (floating points: 32 or 64, access types : 32 bits and task types: 32 bits).

(d) If T is a record type, the specified size must be greater than or equal to the minimal number of bits used to represent values of the type per default.

(e) If T is an array type, the size of the array must be static, i.e. known at compile time and the specified size must be equal to the minimal number of bits used to represent values of the type per default.

Furthermore, the size attribute has effect only if the type is part of a composite type.

```
type BYTE is range 0..255;
   for BYTE'size use 8;
SIXTEEN : BYTE;                    -- one word allocated
EIGHT : array(1..4) of BYTE;       -- one byte per element
```

2. *Collection size specifications:*

   Using the STORAGE_SIZE attribute on an access type will set an upper limit on the total size of objects allocated in the collection allocated for the access type. If further allocation is attempted, the exception STORAGE_ERROR is raised. The specified storage size must be less than or equal to INTEGER'LAST.

3. *Task storage size:*

   When the STORAGE_SIZE attribute is given on a task type, the task stack area will be of the specified size. There is no upper limit on the given size.

4. *Small specifications:*

   Any value of the SMALL attribute less than the specified delta for the fixed point type can be given.

5.3 Enumeration Representation Clauses

Enumeration representation clauses may specify representations in the range of INTEGER'FIRST+1..INTEGER'LAST-1. An enumeration representation clause may be combined with a length clause. If an enumeration representation clause has been given for a type, the representational values are considered when the number of the bits needed to hold any value of the type is evaluated. Thus the type

```
type ENUM is (A, B, C);
for ENUM use (1, 3, 5);
```

needs 3 bits not 2 bits to represent any value of the type ENUM.

5.4 Record Representation Clauses

When component clauses are applied to a record type the following restrictions and interpretations are imposed:

- All values of the component type must be representable within the specified number of bits in the component clause.
- If the component type is either a discrete type, a fixed point type, an array type with a discrete type or a fixed point type as element type, then the component is packed into the specified number of bits (see however the restriction in the paragraph above), and the component may start at any bit boundary.
- If the component type is not one of the types specified in the paragraph above, it must start at a storage unit boundary, a storage unit being 8 bits, and the default size calculated by the compiler must be given as the bit width, i.e. the component must be specified as

  ```
  component at N range 0..16*M-1
  ```

  where `N` specifies the relative storage unit number (0, 1, ...) from the beginning of the record, and `M` the required number of storage units (1, 2, ...).
- The maximum bit width for components of scalar types is 32.

- A record occupies an integral number of storage units.
- A record may take up a maximum of 32Kbits.
- If the component type is an array type with discrete type or a fixed point type as element type, the given bit width must be divisible by the length of the array, i.e. each array element will occupy the same number of bits.

If the record type contains components which are not covered by a component clause, they are allocated consecutively after the component with the value. Allocation of a record component without a component clause is always aligned on a storage unit boundary. Holes created because of component clauses are not otherwise utilized by the compiler.

1. *Alignment Clause*

   Alignment clause for records are implemented with the following characteristics:

   - If the declaration of the record type is done at the outermost level in a library, any alignment is accepted.
   - If the record declaration is done at a given static level (higher than the outermost library level, i.e. the permanent), only word alignments are accepted.

6 Implementation-Dependent Names

None defined by the compiler.

7 Address Clauses

This section describes the implementation of address clause and what types of entities may have their address specified by the user.

7.1 Objects

Address clauses are supported for scalar and composite objects whose size can be determined at compile time. The address value must be static. The given address is the virtual address.

8 Unchecked Conversion

Unchecked conversion is only allowed for types where objects of the same "size". The size of an object is interpreted as follows :

- for arrays, it is the number of storage units occupied be the array elements.
- for records it is the size of the fixed part of the record, i.e. excluding any dynamic storage allocated outside the record.
- for the other non-structured type, the object size is as described in Chapter 9.

For scalar types having a size specification, special rules apply. Conversion involving such a type is allowed if the given size matched either the specified size or the object size.

Example

```
type ACC is access INTEGER;
function TO_INT is new UNCHECKED_CONVERSION(ACC, INTEGER);
                                                    -- OK
function TO_ACC is new
UNCHECKED_CONVERSION(SHORT_INTEGER, ACC, I);
                                                    -- NOT OK
type UNSIGNED is range 0..65535;
for UNISGNED'SIZE use 16;

function TO_INT is new UNCHECKED_CONVERSION(UNSIGNED, INTEGER);
                                                    -- OK
```

```
function TO_SHORT is new
UNCHECKED_CONVERSION(UNSIGNED, SHORT_INTEG_R);
                                                    -- OK
```

End ofexample

9 Input-Output Packages

The implementation supports all requirements of the Ada language and the POSIX standard described in document P1003.5 Draft4.0/WG15-N45. It is an effective interface to the UNIX file system, and in the case of text I/O it is also an effective interface to the UNIX standard input, standard output and standard error streams.

This section describes the functional aspects of the interface to the UNIX file system, including the methods of using the interface to take advantage of the file facilities provided.

The Ada input-output concept as defined in Chapter 14 of the ARM does not constitute a complete functional specification of the input-output packages. Some aspects of I/O system are not discussed at all, while others are intentionally left open for implementation. This section describes those sections not covered in the ARM. Please notice that the POSIX standard puts restrictions on some of the aspects not described in Chapter 14 of the ARM.

The UNIX operating system considers all files to be sequences of characters. Files can either be accessed sequentially or randomly. Files are not structured into records, but an access routine can treat a file as a sequence of records if it arranges the record level input-output.

Note that for sequential or text files (Ada files not UNIX external files) RESET on a file in mode OUT_FILE will empty the file. Also, a sequential or text file opened as an OUT_FILE will be emptied.

9.1 External Files

An external file is either a UNIX disk file, a UNIX FIFO (named pipe), a UNIX pipe, or any device defined in the UNIX directory. The use of devices

such as a tape or communication line may require special access permissions or have restrictions. If an inappropriate operation is attempted on a device, then USE_ERROR exception is raised.

External files created within the UNIX file system shall exist after the termination of the program that created it, and will be accessible from other Ada programs. However, pipes and temporary files will not exist after program termination.

Creation of a file with the same name as an existing external file will cause the existing file to be overwritten.

Creation of files with mode IN_FILE will cause USE_ERROR to be raised.

The name parameter to the input-output routines must be a valid UNIX file name. If the name parameter is empty, then a temporary file is created in the /usr/tmp directory. Temporary files are automatically deleted when they are closed.

9.2 File Management

This section provides useful information for performing file management functions within an Ada program.

1. *Restrictions on Sequential and Direct Input-Output*

   The only restrictions are that placed on the element size, i.e. the number of bytes occupied by the ELEMENT_TYPE : the maximum size allowed is 2147483647 bits; and if the size of the type is variable , the maximum size must be determinable at the point of instantiation from the value of the SIZE attribute for the element type.

2. *The NAME Parameter*

   The NAME Parameter must be a valid UNIX pathname (unless it is the empty string). If any directory in the pathname is inaccessible, USE_ERROR or NAME_ERROR is raised.

   The UNIX names "stdin", "stdout", and "stderr", can be used with TEXT_IO.OPEN. No physical opening of the external file is performed and the Ada file will be associated with the already open external file. These names have no significance for other I/O packages.

Temporary files(NAME = null string) are created using tmpname(3) and will be deleted when CLOSED. Abnormal program termination may leave temporary files in existence. The name function will return the full name of temporary file when it exists.

3. *The Form Parameter*

The Form Parameter, as described below, is applicable to DIRECT_IO, SEQUENTIAL_IO and TEXT_IO operations. The value of the Form Parameter for Ada I/O shall be a character sting. The value of the character string shall be a series of fields separated by commas. Each field shall consist of optional separators, followed by a field name identifier, followed by optional separators, followed by "=>", followed by optional separators, followed by a field value, followed by optional separators. The allowed values for the field names and the corresponding field values are described below. All field names and field values are case-insensitive.

The following BNF describes the syntax of the FORM parameter:

```
form                ::= [field {, field}*]
fields              ::= rights | append |blocking |
                        terminal_input | fifo |
                        posix_file_descriptor
rights              ::= OWNER | GROUP | WORLD =>
access              ::= READ | WRITE | EXECUTE | NONE
access_underscor    ::= _READ | _WRITE | _EXECUTE | _NONE
append              ::= APPEND => YES | NO
blocking            ::= BLOCKING => TASKS | PROGRAM
terminal_input      ::= TERMINAL_INPUT => LINES | CHARACTERS
fifo                ::= FIFO => YES | NO
posix_file_descriptor   ::= POSIX_FILE_DESCRIPTOR => 2
```

The Form Parameter is used to control the following :

(a) File ownership:
Access rights to a file is controlled by the following field names

"OWNER", "GROUP" and "WORLD". The field values are "READ", "WRITE", "EXECUTE" and "NONE" or any combination of the previously listed values separated by underscores. The access rights field names are applicable to TEXT_IO, DIRECT_IO and SEQUENTIAL_IO. The default value is OWNER => READ_WRITE, GROUP => READ_WRITE and WORLD => READ_WRITE. The actual actual access rights on a created file will be the default value subtracted the value of the environment variable umask.

Example
To make a file readable and writable by the owner only, the form Parameter should look something like this:

```
"Owner =>read_write, World=> none, Group=>none"
```

If one or more of the field names are missing the default value is used. The permission field is evaluated in left-to-right order. An ambiguity may arise with a Form Parameter of the following:

```
"Owner=>Read_Execute_None_Write_Read"
```

In this instance, using the left-to-right evaluation order, the "None" field will essentially reset the permissions to none and this example would have the access rights WRITE and READ.

(b) Appending to a file:
Appending to a file is achieved by using field name "APPEND" and one of the two field values "YES" or "NO". The default value is "NO". "Append" is allowed with both TEXT_IO and SEQUENTIAL_IO. The effect of appending to a file that all output to that file is written to the end of the named external file. This field may only be used with the "OPEN" operation, using the field name "APPEND" in connection with a "CREATE" operation shall raise USE_ERROR. Furthermore, a USE_ERROR is raised if the specified file is a terminal device or another device.

Example
To append to a file, one would write:

```
"Append => Yes"
```

(c) Blocking vs. non-blocking I/O:
The blocking field name is "Blocking" and the field values are "TASKS" and "PROGRAM". The default value is "PROGRAM". "Blocking=>Tasks" causes the calling task, but no others, to wait for the completion of an I/O operation. "Blocking=>program" causes all tasks within the program to wait for the completion of the I/O operation. The blocking mechanism is applicable to TEXT_IO, DIRECT_IO and SEQUENTIAL_IO. UNIX does not allow the support of "BLOCKING=>TASKS" currently.

(d) How characters are read from the keyboard:
The field name is "TERMINAL_INPUT" and field value is either "LINES" or "CHARACTERS". The effect of the field value "Terminal_input => Characters" is that characters are read in a noncanonical fashion with Minimum_count=1, meaning one character at a time=0.0 corresponding so that the read operation is not satisfied until Minimum_Count characters are received. If field value "LINES" is used, the characters are read one line at a time in canonical mode. The default value is Lines. "TERMINAL_INPUT" has effect if the specified file is not already open or if the file is not open on a terminal. It is permitted for the same terminal device to be opened for input in both modes as separate Ada file objects. In this case, no user input characters shall be read from the input device without an explicit input operation on one of the file objects. The "TERMINAL_INPUT" mechanism is only applicable to TEXT_IO.

(e) Creation of FIFO files:
The field name is "FIFO" and the field value is either "YES" or "NO". "FIFO => YES" means that the file shall be a named FIFO file. The default value is "NO". For use with TEXT_IO, the "FIFO" field is only allowed with the CREATE operation. If used in connection with an OPEN operation, USE_ERROR is raised.

For SEQUENTIAL_IO, the FIFO mechanism is applicable for both the CREATE and OPEN operation.

In connection with SEQUENTIAL_IO, an additional field name "O_NDELAY" is used. The field values allowed for "O_NDELAY" are "YES" and "NO". Default is "NO". The "O_NDELAY" field name is provided to allow waiting or immediate return. If, for example, the following form parameter is given:

```
"FIFO=>Yes, O_NDELAY=>Yes"
```

then waiting is performed until completion of the operation. The "O_Ndelay" field name only has meaning in connection with the FIFO facility and otherwise ignored.

(f) Access to open POSIX files:
The field name is "POSIX_File_Descriptor". The field value is the character string "2" which denotes the stderr file. Any other field value will result in USE_ERROR being raised. The NAME parameter provides the value which will be returned by subsequent usage of the NAME function. The operation does not change the state of the file. During the period that the Ada file is open, the result of any file operations on the file descriptor are undefined. Note that this is a method to make stderr accessible from an Ada program.

4. *File Access*

The following guidelines should be observed when performing file I/O operations:

- At a given instant, any number of files in an Ada program can be associated with corresponding external files.
- When sharing files between programs, it is the responsibility of the programmer to determine the effects of sharing files.
- The RESET and OPEN operations to files with mode OUT_FILE will empty the contents of the file in SEQUENTIAL_IO and TEXT_IO.
- Files can be interchanged between SEQUENTIAL_IO and DIRECT_IO without any special operations if the files are of the same object type.

9.3 Buffering

The Ada I/O system provides buffering in addition to the buffering provided by UNIX. The Ada TEXT_IO packages will flush all output to the operating system under the following circumstances:

1. The device is a terminal device and an end of line, end of page, or end of file has occurred.

2. The device is a terminal device and the same Ada program makes an Ada TEXT_IO input request or another file object representing the same device.

9.4 Package IO_EXCEPTIONS

The specification of package IO_EXCEPTIONS:

```
Package IO_EXCEPTIONS is

-- The order of the following declarations must NOT be changed:

STATUS_ERROR    : exception;
MODE_ERROR      : exception;
NAME_ERROR      : exception;
USE_ERROR       : exception;
DEVICE_ERROR    : exception;
END_ERROR       : exception;
DATA_ERROR      : exception;
LAYOUT_ERROR    : exception;

end IO_EXCEPTIONS;
```

9.5 Sequential Input-Output

The implementation omits type checking for DATA_ERROR, in case the element type is of an unconstrained type, ARM 14.2.2(4), i.e.:

```
...f : FILE_TYPE
type et  is 1..100;
type eat is array( et range <> ) of integer;
X : eat( 1..2 );
Y : eat( 1..4 );
...
-- write X, Y:
write( f, X); write( f, Y); reset( f, IN_FILE);
-- read X into Y and Y into X:
read( f, Y); read( f, X);
```

This will give undefined values in the last 2 elements of Y, and not DATA_ERROR.

1. *Specification of the Package Sequential IO*

```
with BASIC_IO_TYPES;
with IO_EXCEPTIONS;
generic
   type ELEMENT_TYPE is private;
package SEQUENTIAL_IO is
   type FILE_TYPE is limited private;
   type FILE_MODE is (IN_FILE, OUT_FILE);

-- File management
   procedure CREATE(FILE : in out FILE_TYPE;
                    MODE : in     FILE_MODE := OUT_FILE;
                    NAME : in     STRING    := "";
                    FORM : in     STRING    := "");
   procedure OPEN  (FILE : in out FILE_TYPE;
                    MODE : in     FILE_MODE;
                    NAME : in     STRING;
                    FORM : in     STRING := "");
   procedure CLOSE (FILE : in out FILE_TYPE);
```

```
   procedure DELETE(FILE : in out FILE_TYPE);
   procedure RESET (FILE : in out FILE_TYPE;
                    MODE : in     FILE_MODE);
   procedure RESET (FILE : in out FILE_TYPE);
   function MODE   (FILE : in FILE_TYPE) return FILE_MODE;
   function NAME   (FILE : in FILE_TYPE) return STRING;
   function FORM   (FILE : in FILE_TYPE) return STRING;
   function IS_OPEN(FILE : in FILE_TYPE) return BOOLEAN;

-- input and output operations
   procedure READ  (FILE : in     FILE_TYPE;
                    ITEM :    out ELEMENT_TYPE);
   procedure WRITE (FILE : in FILE_TYPE;
                    ITEM : in ELEMENT_TYPE);
   function END_OF_FILE(FILE : in FILE_TYPE) return BOOLEAN;

-- exceptions
   STATUS_ERROR : exception renames IO_EXCEPTIONS.STATUS_ERROR;
   MODE_ERROR   : exception renames IO_EXCEPTIONS.MODE_ERROR;
   NAME_ERROR   : exception renames IO_EXCEPTIONS.NAME_ERROR;
   USE_ERROR    : exception renames IO_EXCEPTIONS.USE_ERROR;
   DEVICE_ERROR : exception renames IO_EXCEPTIONS.DEVICE_ERROR;
   END_ERROR    : exception renames IO_EXCEPTIONS.END_ERROR;
   DATA_ERROR   : exception renames IO_EXCEPTIONS.DATA_ERROR;

private
   type FILE_TYPE is new BASIC_IO_TYPES.FILE_TYPE;
end SEQUENTIAL_IO;
```

9.6 Direct Input-Output

The implementation omits type checking for DATA_ERROR, in case the element type is of an unconstrained type.

1. *Specification of the Package Direct IO*

```
with BASIC_IO_TYPES;
with IO_EXCEPTIONS;
generic
   type ELEMENT_TYPE is private;
package DIRECT_IO is
   type FILE_TYPE is limited private;
   type FILE_MODE is (IN_FILE, INOUT_FILE, OUT_FILE);
   type COUNT is range 0..INTEGER'LAST;
   subtype POSITIVE_COUNT is COUNT range 1..COUNT'LAST;

-- File management
   procedure CREATE(FILE : in out FILE_TYPE;
                    MODE : in     FILE_MODE  := INOUT_FILE;
                    NAME : in     STRING     := "";
                    FORM : in     STRING     := "");
   procedure OPEN  (FILE : in out FILE_TYPE;
                    MODE : in     FILE_MODE;
                    NAME : in     STRING;
                    FORM : in     STRING     := "");
   procedure CLOSE (FILE : in out FILE_TYPE);
   procedure DELETE(FILE : in out FILE_TYPE);
   procedure RESET (FILE : in out FILE_TYPE;
                    MODE : in     FILE_MODE);
   procedure RESET (FILE : in out FILE_TYPE);
   function MODE   (FILE : in FILE_TYPE) return FILE_MODE;
   function NAME   (FILE : in FILE_TYPE) return STRING;
   function FORM   (FILE : in FILE_TYPE) return STRING;
   function IS_OPEN(FILE : in FILE_TYPE) return BOOLEAN;

-- input and output operations
   procedure READ  (FILE : in     FILE_TYPE;
                    ITEM :    out ELEMENT_TYPE;
                    FROM : in     POSITIVE_COUNT);
   procedure READ  (FILE : in     FILE_TYPE;
                    ITEM :    out ELEMENT_TYPE);
   procedure WRITE (FILE : in FILE_TYPE;
```

```
                        ITEM : in ELEMENT_TYPE;
                        TO   : in POSITIVE_COUNT);
      procedure WRITE (FILE : in FILE_TYPE;
                       ITEM : in ELEMENT_TYPE);
      procedure SET_INDEX(FILE : in FILE_TYPE;
                          TO   : in POSITIVE_COUNT);
      function INDEX(FILE : in FILE_TYPE) return POSITIVE_COUNT;
      function SIZE (FILE : in FILE_TYPE) return COUNT;
      function END_OF_FILE(FILE : in FILE_TYPE) return BOOLEAN;

   -- exceptions
      STATUS_ERROR : exception renames IO_EXCPTIONS.STATUS_ERROR;
      MODE_ERROR   : exception renames IO_EXCPTIONS.MODE_ERROR;
      NAME_ERROR   : exception renames IO_EXCPTIONS.NAME_ERROR;
      USE_ERROR    : exception renames IO_EXCPTIONS.USE_ERROR;
      DEVICE_ERROR : exception renames IO_EXCPTIONS.DEVICE_ERROR;
      END_ERROR    : exception renames IO_EXCPTIONS.END_ERROR;
      DATA_ERROR   : exception renames IO_EXCPTIONS.DATA_ERROR;

   private
     type FILE_TYPE is new BASIC_IO_TYPES.FILE_TYPE;
   end DIRECT_IO;
```

9.7 Specification of the Package Text_IO

```
with BASIC_IO_TYPES;
with IO_EXCEPTIONS;
package TEXT_IO is
   type FILE_TYPE is limited private;
   type FILE_MODE is (IN_FILE, OUT_FILE);
   type    COUNT is range 0 .. INTEGER'LAST;
   subtype POSITIVE_COUNT is COUNT range 1 .. COUNT'LAST;
   UNBOUNDED: constant COUNT:= 0; -- line and page length
   -- max. size of an integer output field 2#....#
   subtype FIELD          is INTEGER range 0 .. 35;
```

```
    subtype NUMBER_BASE    is INTEGER range 2 .. 16;
    type TYPE_SET is (LOWER_CASE, UPPER_CASE);

-- File Management
    procedure CREATE (FILE  : in out FILE_TYPE;
                      MODE  : in     FILE_MODE := OUT_FILE;
                      NAME  : in     STRING    := "";
                      FORM  : in     STRING    := "");
    procedure OPEN   (FILE  : in out FILE_TYPE;
                      MODE  : in     FILE_MODE;
                      NAME  : in     STRING;
                      FORM  : in     STRING    := "");
    procedure CLOSE  (FILE  : in out FILE_TYPE);
    procedure DELETE (FILE  : in out FILE_TYPE);
    procedure RESET  (FILE  : in out FILE_TYPE;
                      MODE  : in     FILE_MODE);
    procedure RESET  (FILE  : in out FILE_TYPE);
    function  MODE   (FILE  : in FILE_TYPE) return FILE_MODE;
    function  NAME   (FILE  : in FILE_TYPE) return STRING;
    function  FORM   (FILE  : in FILE_TYPE) return STRING;
    function  IS_OPEN(FILE  : in FILE_TYPE) return BOOLEAN;

    -- Control of default input and output files
    procedure  SET_INPUT    (FILE : in FILE_TYPE);
    procedure  SET_OUTPUT   (FILE : in FILE_TYPE);
    function   STANDARD_INPUT    return FILE_TYPE;
    function   STANDARD_OUTPUT   return FILE_TYPE;
    function   CURRENT_INPUT     return FILE_TYPE;
    function   CURRENT_OUTPUT    return FILE_TYPE;

    -- specification of line and page lengths
    procedure SET_LINE_LENGTH  (FILE : in FILE_TYPE;
                                TO   : in COUNT);
    procedure SET_LINE_LENGTH  (TO   : in COUNT);
    procedure SET_PAGE_LENGTH  (FILE : in FILE_TYPE;
                                TO   : in COUNT);
    procedure SET_PAGE_LENGTH  (TO   : in COUNT);
```

```
function  LINE_LENGTH      (FILE : in FILE_TYPE) return COUNT;
function  LINE_LENGTH                            return COUNT;
function  PAGE_LENGTH      (FILE : in FILE_TYPE) return COUNT;
function  PAGE_LENGTH                            return COUNT;

-- Column, Line, and Page Control
procedure NEW_LINE    (FILE    : in FILE_TYPE;
                       SPACING : in POSITIVE_COUNT := 1);
procedure NEW_LINE    (SPACING : in POSITIVE_COUNT := 1);
procedure SKIP_LINE   (FILE    : in FILE_TYPE;
                       SPACING : in POSITIVE_COUNT := 1);
procedure SKIP_LINE   (SPACING : in POSITIVE_COUNT := 1);
function  END_OF_LINE (FILE : in FILE_TYPE) return BOOLEAN;
function  END_OF_LINE                       return BOOLEAN;
procedure NEW_PAGE    (FILE : in FILE_TYPE);
procedure NEW_PAGE                          ;
procedure SKIP_PAGE   (FILE : in FILE_TYPE);
procedure SKIP_PAGE                         ;
function  END_OF_PAGE (FILE : in FILE_TYPE) return BOOLEAN;
function  END_OF_PAGE                       return BOOLEAN;
function  END_OF_FILE (FILE : in FILE_TYPE) return BOOLEAN;
function  END_OF_FILE                       return BOOLEAN;
procedure SET_COL     (FILE : in FILE_TYPE;
                       TO   : in POSITIVE_COUNT);
procedure SET_COL     (TO   : in POSITIVE_COUNT);
procedure SET_LINE    (FILE : in FILE_TYPE;
                       TO   : in POSITIVE_COUNT);
procedure SET_LINE    (TO   : in POSITIVE_COUNT);
function  COL         (FILE : in FILE_TYPE)
                                  return POSITIVE_COUNT;
function  COL                     return POSITIVE_COUNT;
function  LINE        (FILE : in FILE_TYPE)
                                  return POSITIVE_COUNT;
function  LINE                    return POSITIVE_COUNT;
function  PAGE        (FILE : in FILE_TYPE)
                                  return POSITIVE_COUNT;
function  PAGE                    return POSITIVE_COUNT;
```

```
    -- Character Input-Output
    procedure GET  (FILE : in     FILE_TYPE;
                    ITEM :    out CHARACTER);
    procedure GET  (ITEM :    out CHARACTER);
    procedure PUT  (FILE : in FILE_TYPE;
                    ITEM : in CHARACTER);
    procedure PUT  (ITEM : in CHARACTER);
    -- String Input-Output

    procedure GET  (FILE : in     FILE_TYPE;
                    ITEM :    out STRING);
    procedure GET  (ITEM :    out STRING);
    procedure PUT  (FILE : in FILE_TYPE;
                    ITEM : in STRING);
    procedure PUT  (ITEM : in STRING);
    procedure GET_LINE  (FILE : in     FILE_TYPE;
                         ITEM :    out STRING;
                         LAST :    out NATURAL);
    procedure GET_LINE  (ITEM :    out STRING;
                         LAST :    out NATURAL);
    procedure PUT_LINE  (FILE : in     FILE_TYPE;
                         ITEM : in     STRING);
    procedure PUT_LINE  (ITEM : in     STRING);

-- Generic Package for Input-Output of Integer Types
  generic
      type NUM is range <>;
  package INTEGER_IO is
     DEFAULT_WIDTH : FIELD        := NUM'WIDTH;
     DEFAULT_BASE  : NUMBER_BASE  :=        10;
     procedure GET  (FILE  : in     FILE_TYPE;
                     ITEM  :    out NUM;
                     WIDTH : in     FIELD := 0);
     procedure GET  (ITEM  :    out NUM;
                     WIDTH : in     FIELD := 0);
     procedure PUT  (FILE  : in FILE_TYPE;
```

```
                    ITEM  : in NUM;
                    WIDTH : in FIELD := DEFAULT_WIDTH;
                    BASE  : in NUMBER_BASE := DEFAULT_BASE);
    procedure PUT  (ITEM  : in NUM;
                    WIDTH : in FIELD := DEFAULT_WIDTH;
                    BASE  : in NUMBER_BASE := DEFAULT_BASE);
    procedure GET  (FROM  : in     STRING;
                    ITEM  :    out NUM;
                    LAST  :    out POSITIVE);
    procedure PUT  (TO    :    out STRING;
                    ITEM  : in     NUM;
                    BASE  : in     NUMBER_BASE := DEFAULT_BASE);
  end INTEGER_IO;

-- Generic Packages for Input-Output of Real Types
  generic
    type NUM is digits <>;
  package FLOAT_IO is
    DEFAULT_FORE : FIELD :=             2;
    DEFAULT_AFT  : FIELD := NUM'digits - 1;
    DEFAULT_EXP  : FIELD :=             3;
    procedure GET  (FILE  : in     FILE_TYPE;
                    ITEM  :    out NUM;
                    WIDTH : in     FIELD := 0);
    procedure GET  (ITEM  :    out NUM;
                    WIDTH : in     FIELD := 0);
    procedure PUT  (FILE : in FILE_TYPE;
                    ITEM : in NUM;
                    FORE : in FIELD := DEFAULT_FORE;
                    AFT  : in FIELD := DEFAULT_AFT;
                    EXP  : in FIELD := DEFAULT_EXP);
    procedure PUT  (ITEM : in NUM;
                    FORE : in FIELD := DEFAULT_FORE;
                    AFT  : in FIELD := DEFAULT_AFT;
                    EXP  : in FIELD := DEFAULT_EXP);
    procedure GET  (FROM : in     STRING;
                    ITEM :    out NUM;
```

```
                    LAST :    out POSITIVE);
    procedure PUT  (TO   :    out STRING;
                    ITEM : in     NUM;
                    AFT  : in     FIELD := DEFAULT_AFT;
                    EXP  : in     FIELD := DEFAULT_EXP);
  end FLOAT_IO;

  generic
    type NUM is delta <>;
  package FIXED_IO is
    DEFAULT_FORE : FIELD := NUM'FORE;
    DEFAULT_AFT  : FIELD := NUM'AFT;
    DEFAULT_EXP  : FIELD := 0;
    procedure GET  (FILE  : in     FILE_TYPE;
                    ITEM  :    out NUM;
                    WIDTH : in     FIELD := 0);
    procedure GET  (ITEM  :    out NUM;
                    WIDTH : in     FIELD := 0);
    procedure PUT  (FILE : in FILE_TYPE;
                    ITEM : in NUM;
                    FORE : in FIELD := DEFAULT_FORE;
                    AFT  : in FIELD := DEFAULT_AFT;
                    EXP  : in FIELD := DEFAULT_EXP);
    procedure PUT  (ITEM : in NUM;
                    FORE : in FIELD := DEFAULT_FORE;
                    AFT  : in FIELD := DEFAULT_AFT;
                    EXP  : in FIELD := DEFAULT_EXP);
    procedure GET  (FROM : in     STRING;
                    ITEM :    out NUM;
                    LAST :    out POSITIVE);
    procedure PUT  (TO   :    out STRING;
                    ITEM : in     NUM;
                    AFT  : in     FIELD := DEFAULT_AFT;
                    EXP  : in     FIELD := DEFAULT_EXP);
  end FIXED_IO;

-- Generic Package for Input-Output of Enumeration Types
```

```
generic
   type ENUM is (<>);
package ENUMERATION_IO is
   DEFAULT_WIDTH   : FIELD    := 0;
   DEFAULT_SETTING : TYPE_SET := UPPER_CASE;
   procedure GET  (FILE : in     FILE_TYPE;
                   ITEM :    out ENUM);
   procedure GET  (ITEM :    out ENUM);
   procedure PUT  (FILE  : in FILE_TYPE;
                   ITEM  : in ENUM;
                   WIDTH : in FIELD      := DEFAULT_WIDTH;
                   SET   : in TYPE_SET   := DEFAULT_SETTING);
   procedure PUT  (ITEM  : in ENUM;
                   WIDTH : in FIELD      := DEFAULT_WIDTH;
                   SET   : in TYPE_SET   := DEFAULT_SETTING);
   procedure GET  (FROM : in     STRING;
                   ITEM :    out ENUM;
                   LAST :    out POSITIVE);
   procedure PUT  (TO   :    out STRING;
                   ITEM : in     ENUM;
                   SET  : in     TYPE_SET := DEFAULT_SETTING);
end ENUMERATION_IO;

   -- Exceptions
   STATUS_ERROR : exception renames IO_EXCEPTIONS.STATUS_ERROR;
   MODE_ERROR   : exception renames IO_EXCEPTIONS.MODE_ERROR;
   NAME_ERROR   : exception renames IO_EXCEPTIONS.NAME_ERROR;
   USE_ERROR    : exception renames IO_EXCEPTIONS.USE_ERROR;
   DEVICE_ERROR : exception renames IO_EXCEPTIONS.DEVICE_ERROR;
   END_ERROR    : exception renames IO_EXCEPTIONS.END_ERROR;
   DATA_ERROR   : exception renames IO_EXCEPTIONS.DATA_ERROR;
   LAYOUT_ERROR : exception renames IO_EXCEPTIONS.LAYOUT_ERROR;
private
  type FILE_BLOCK_TYPE is new BASIC_IO_TYPES.FILE_TYPE;
  type FILE_OBJECT_TYPE is record
  IS_OPEN    : BOOLEAN    := FALSE;
  FILE_BLOCK : FILE_BLOCK_TYPE;
```

```
    end record;
    type FILE_TYPE is access FILE_OBJECT_TYPE;
end TEXT_IO;
```

9.8 Low Level Input-Output

The package LOW_LEVEL_IO is empty.

9.9 Package TERMINAL

The specification of package TERMINAL:

```
with COMMON_DEFS;
use  COMMON_DEFS;
Package TERMINAL is
procedure SET_CURSOR(ROW, COL     : in     INTEGER);
procedure IN_CHARACTER(CH    :   out CHARACTER);
procedure IN_INTEGER  (I     :   out INTEGER);
procedure IN_LINE     (T     :   out TERMINAL_LINE);
procedure OUT_CHARACTER(CH   : in    CHARACTER);
procedure OUT_INTEGER (I     : in    INTEGER);
procedure OUT_INTEGER_F(I, W         : in     INTEGER);
procedure OUT_LINE     (L    : in    STRING);
procedure OUT_STRING   (S    : in    STRING);
procedure OUT_NL;
procedure OUT_FF;
procedure FLUSH;
procedure OPEN_LOG_FILE(FILE_NAME    : in STRING);
procedure CLOSE_LOG_FILE;
end TERMINAL;
```